UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ELETRÔNICA

Projeto Didático : Sistema de Controle de Estoque de Supermercado em Ambiente LINUX

Autor : *Rodrigo de Carvalho Roman*

Orientador : ______________________________
Prof. Aloysio de Castro Pinto Pedroza

Examinador : ______________________________
Prof. Jorge Lopes de Souza Leão

Examinador : ______________________________
Prof. Marcelo Luiz Drumond Lanza

DEL
Novembro de 1998

// Resumo

Este projeto final tem, como objetivo principal, apresentar uma ferramenta que pode ser útil na familiarização e aprendizado dos sistemas operacionais multitarefa.

Basicamente resolveremos o problema de automação do controle de estoque de um supermercado em tempo real utilizando um sistema operacional multitarefa. O termo "tempo real" se aplica à situação de baixa no estoque interno do estabelecimento (produtos disponíveis nas gôndolas) que será realizada conforme os produtos forem registrados nos caixas. Apesar de hipotético este problema é perfeitamente possível de se enquadrar em uma situação real.

Neste projeto utilizaremos o LINUX, que é um sistema operacional que executa em PCs e que é do tipo UNIX, duas das vantagens principais que nos levaram a fazer esta escolha. Além disso optamos por uma implementação na linguagem ANSI C, que é amplamente conhecida e utilizada.

Os programas desenvolvidos estão divididos em duas categorias principais. Na categoria EXEMPLOS temos pequenos programas, muito bem documentados, que mostram como implementar as principais facilidades que serão utilizadas na nossa solução. Os conceitos mais importantes que abordaremos tratarão de PROCESSOS (onde incluímos operações e comunicação por troca de mensagens utilizando SOCKETS) e de acesso disciplinado à regiões compartilhadas de dados com exclusão mútua (onde incluímos SEMÁFOROS). Todos os programas e os conceitos envolvidos serão discutidos de maneira apropriada.

Já os programas da categoria ESTUDO DE CASO implementam a solução para o nosso problema. Estes programas apresentam uma combinação das técnicas descritas nos programas da categoria de EXEMPLOS com as complementações necessárias. Nesta categoria notamos que os programas possuem razoável extensão. Para cada função introduzimos comentários nos próprios códigos, a fim de que isto possa ajudar o leitor na análise. Estes programas também serão comentados e discutidos.

Procuramos, dentro do possível, tornar estes programas portáveis para outros sistemas multitarefa do tipo UNIX que executem em PCs. Os comentários serão direcionados para a execução também sobre o FreeBSD, enfatizando as mudanças que são necessárias para que isto seja possível.

Sendo assim colocamos à disposição dos leitores todos os programas desenvolvidos e acreditamos que este trabalho possa ter utilidade para os interessados ou como ferramenta de apoio à alguma cadeira acadêmica que trate deste assunto.

Palavras-chave

1. Sistemas operacionais
2. Multitarefa
3. Processo
4. Comunicação
5. Regiões críticas

Índice

Introdução

1.1) - Objetivos

O objetivo deste projeto final é a elaboração de uma ferramenta de apoio ao aprendizado e familiarização com os sistemas operacionais. Os textos didáticos apresentados em alguns livros que tratam do assunto são direcionados para uma abordagem conceitual e bastante abrangente. Isto se deve à grande variedade de sistemas operacionais desenvolvidos e de suas diversas versões disponíveis no mercado. O que se pretende com este trabalho é a exemplificação prática, de maneira didática, de um subconjunto destes conceitos teóricos e a utilização dos mesmos na solução de um problema real utilizando um sistema operacional particular. O sistema operacional escolhido como ambiente de desenvolvimento foi o LINUX. Esta escolha deveu-se a dois fatores principais :

i) - O LINUX pode ser executado em qualquer computador da linha PC, desde que se respeite uma configuração mínima bastante acessível e comum (no mínimo um 386). Desta maneira os usuários podem instalar o LINUX em sua residência ou máquina pessoal, utilizar e analisar todos os programas desenvolvidos neste projeto;

ii) - O LINUX é um sistema operacional do tipo UNIX. Os sistemas do tipo UNIX são, atualmente, sistemas de destacada importância principalmente devido a sua larga utilização e eficiência como sistemas multitarefa. De uma maneira geral podemos afirmar que o estudo do LINUX oferece base suficiente para utilização de outros sistemas do mesmo tipo. Desta forma teríamos uma relativa portabilidade dos exemplos desenvolvidos, que poderiam ser executados em outros sistemas UNIX, efetuando-se algumas mudanças simples (quando necessário). Além disso o estudo de um sistema com estas características pode ser proveitoso para o leitor.

Dentro do LINUX escolhemos uma implementação em linguagem ANSI C, cujo compilador já vem embutido na instalação de qualquer sistema UNIX. Mesmo que o leitor não tenha muitos conhecimentos em C, a idéia é que os programas possam ser compreendidos de uma maneira global. Além disso há alguma semelhança entre a linguagem ANSI C e o PASCAL, que é uma linguagem de programação bastante popular no meio acadêmico dentro das áreas de ciências exatas.

Desta forma, foram desenvolvidos diversos programas pertencentes a uma dentre duas possíveis categorias básicas a definir :

1) - Categoria EXEMPLOS ;

2) - Categoria ESTUDO DE CASO.

Na categoria EXEMPLOS temos programas simples, detalhadamente documentados, que mostram didaticamente a implementação prática de conceitos estudados na teoria e que serão utilizados intensamente na solução do problema maior em questão. Estes programas devem ser analisados e executados para que se possa entender o seu objetivo. Eles serão a fonte de referência para o desenvolvimento de futuros projetos práticos. Entretanto, dependendo da aplicação, é provável que o leitor tenha que desenvolver outros programas abordando conceitos que não foram utilizados para a nossa solução. Basicamente estes exemplos tratam de *OPERAÇÕES COM PROCESSOS, COMUNICAÇÃO ENTRE PROCESSOS POR TROCA DE MENSAGENS UTILIZANDO SOCKETS e ACESSO CONCORRIDO À REGIÕES COMPARTILHADAS DE DADOS UTILIZANDO SEMÁFOROS.*

Na categoria ESTUDO DE CASO temos todas as bibliotecas e programas desenvolvidos para a solução do problema de *AUTOMAÇÃO DE CONTROLE DE ESTOQUE EM TEMPO REAL DE UM SUPERMERCADO*. Dentro destes programas utilizamos, entre outras rotinas, partes dos programas da categoria EXEMPLOS. Estes programas mostram como utilizar as facilidades exemplificadas para resolver uma situação mais complexa.

Todos estes programas foram desenvolvidos e testados sobre o LINUX e sobre o FreeBSD. Mais detalhes serão abordados no anexo que trata da portabilidade dos programas.

O microcomputador utilizado para nosso desenvolvimento foi um PC 486 INTEL com 8Mb de memória RAM. É perfeitamente possível que se execute todos estes programas em um computador caseiro, desde que se instale uma versão atual do LINUX ou FreeBSD. A instalação do X-WINDOWS não é condição necessária para a execução dos mesmos. No anexo que trata da instalação do LINUX pode ser encontrado um guia de referência importante com algumas informações e dicas para quem deseja ter e executar em seu computador pessoal estes programas.

1.2) - Roteiro da documentação

Vamos apresentar, a seguir, um roteiro dos capítulos subseqüentes desta documentação apresentando, resumidamente, os assuntos que serão tratados em cada um deles.

No capítulo 2 serão abordados conceitos teóricos que serão utilizados na solução do problema maior em questão. Estes conceitos são implementados no LINUX através dos programas da categoria

EXEMPLOS. Após a definição de determinado conceito temos o comentário do programa relativo a sua implementação.

No capítulo 3 temos a descrição e a modelagem do problema de automação de controle de estoque de um supermercado, baseada nos conceitos abordados e exemplificados no capítulo 2. Utilizaremos para a nossa modelagem as REDES DE PETRI e validaremos a modelagem através de uma ferramenta bastante útil e que foi desenvolvida especificamente para este propósito.

No capítulo 4 iremos comentar os programas que foram desenvolvidos para solucionar o problema que será modelado no capítulo 3. Cada programa desenvolvido será comentado em separado. Ao todo são sete programas a saber : CAIXA, PTA (Processo para Tratamento de Atualizações), enviaTP, PAI (Processo Atualizador de Itens), SUPERVISOR, GERENTE e PAR (Processo para Aviso de Reposição). Alguns destes programas são bastante extensos, de modo que não comentaremos cada linha de código. Os trechos mais importantes destes programas serão apresentados e comentados. Além disso eles contêm documentação no próprio código, o que também pode ser de utilidade para uma melhor compreensão dos mesmos. No final deste capítulo mostramos alguns resultados da implementação.

No capítulo 5 procederemos nos comentários e tiraremos as conclusões finais acerca do que foi realizado, comentando algumas dificuldades de projeto. Também serão propostos novos temas para trabalhos futuros que poderão dar continuidade a este projeto.

Com o intuito de esclarecer alguns pontos adicionais importantes serão apresentados cinco anexos.

O anexo A apresentará um guia de dicas rápido e importante para as pessoas que pretendem instalar o LINUX em seu computador pessoal.

O anexo B especificará as mudanças necessárias para que se possa rodar os programas em seu computador pessoal e as diferenças na maneira de se compilar estes programas entre o LINUX e o FreeBSD.

O anexo C descreverá como consultar e executar todos os programas que foram desenvolvidos.

O anexo D apresentará as listagens de todos os programas que pertencem à categoria EXEMPLOS e que são comentados no capítulo 2.

Para um perfeito entendimento do contexto do problema e da solução é fortemente aconselhável que se leia todas as seções desta documentação. Entretanto, se o leitor acredita ter intimidade com os conceitos teóricos que são abordados e exemplificados no capítulo 2, pode apenas folhear ou mesmo transpassar este capítulo.

Capítulo 2 - O sistema operacional LINUX

2.1) - Introdução

O sistema operacional LINUX é um sistema multitarefa de grande eficiência. A sua utilização em diversos segmentos do meio profissional, na atualidade, é um fato.

O estudo teórico acerca de conceitos que podem ser úteis na solução de um problema real é um passo importante. Para o nosso problema hipotético será proposta, posteriormente, uma solução totalmente baseada na característica de multiprocessamento do sistema LINUX. Vamos iniciar este projeto pelo estudo de alguns conceitos relacionados ao sistema LINUX. Os conceitos podem e devem ser gerais; implementações devem ser particulares (embora se procure a portabilidade delas). Após a abordagem de determinado conceito deve-se conhecer o modo pelo qual podemos utilizá-lo, a nível prático, no ambiente em que estamos trabalhando. Com este intuito será apresentada uma série de programas em linguagem ANSI C, os quais serão comentados neste capítulo. Quando necessário, trechos do código serão evidenciados para auxiliar nesta tarefa. Além disso estes programas contêm documentação em forma de comentários no próprio código, o que pode ser de grande utilidade ao se visualizar o mesmo. Os conceitos que aqui serão abordados podem ser encontrados em qualquer livro que trate de sistemas operacionais. Na verdade, os textos encontrados nestes livros podem se mostrar muito mais completos e esclarecedores do que o nosso. Não é nosso objetivo, neste capítulo, fornecer toda a base teórica para o leitor. Os conceitos que serão comentados neste capítulo são básicos e bastante simples. Nem todos os assuntos abordados aqui terão, obrigatoriamente, alguma implementação prática. Em certos casos citaremos superficialmente métodos diferentes para uma mesma implementação apenas a título ilustrativo. Apenas os conceitos que serão de fato utilizados terão implementação prática.

Na seção 2.2 abordaremos o conceito de "system call" de um sistema operacional. Para este conceito não teremos exemplo específico (as chamadas de sistema serão utilizadas em todos os programas).

Na seção 2.3 abordaremos conceitos relativos a "processos". Temos exemplos que implementam duplicação e triplicação de processos utilizando a primitiva "FORK ( )", lançamento de processos em "BACKGROUND" utilizando a primitiva "EXEC ( )" e finalização de processos utilizando a primitiva "KILL ( )".

Na seção 2.4 lançaremos o conceito de comunicação entre processos utilizando troca de mensagens. O método que será utilizado posteriormente na nossa solução é baseado na utilização dos SOCKETS. Para exemplificar esta situação desenvolvemos programas que implementam uma arquitetura CLIENTE-SERVIDOR com acesso concorrido (de vários clientes). Estes programas mostram os procedimentos e primitivas utilizadas para o estabelecimento de uma conexão e para a transmissão de dados em uma rede, assim como o formato das mensagens utilizadas.

Na seção 2.5 trataremos do acesso disciplinado, com exclusão mútua, à regiões compartilhadas de dados utilizando semáforos como forma de comunicação entre processos. Para este tópico apresentaremos um programa exemplo que será devidamente comentado. Neste programa teremos as rotinas que implementam as primitivas "UP ( )", "DOWN ( )" e "INITSEM ( )".

2.2) - "System calls" de um sistema operacional

As chamadas de sistema ("system calls") comunicam os programas do usuário e o sistema operacional, requisitando serviços deste último. Para cada chamada de sistema existe um procedimento associado que pode ser acessado pelos programas do usuário. Este procedimento guarda os parâmetros passados na chamada em algum lugar determinado (por exemplo registradores) e dispara uma instrução TRAP (um tipo de chamada de procedimento protegida, também chamada de primitiva) para ativar o sistema operacional. A palavra "primitiva" nasceu do conceito de indivisível. Uma vez acionada uma primitiva de um sistema operacional, o usuário que fez a chamada deve, normalmente, aguardar o resultado. Estes procedimentos têm a função de esconder os detalhes da instrução TRAP e fazer com que as chamadas de sistema pareçam com chamadas de procedimentos comuns.

O sistema operacional, após o TRAP, examina os parâmetros passados para saber se são válidos ou não, para que possa realizar o pedido. Após a finalização da tarefa o sistema operacional coloca um código de "status" em um registrador, para indicar se a operação foi concluída com sucesso ou não. O procedimento retorna, para quem fez a chamada, esta informação na maneira usual, como um valor de função.

Este conceito pode ser um pouco confuso. Suponha que dentro de um programa fazemos uma chamada da função "write". Ao se fazer a chamada de procedimento (que pode ser utilizada pelo usuário) os argumentos passados são armazenados em algum local e há a chamada da primitiva "WRITE" associada. O sistema retornará um valor para indicar se a operação foi ou não concluída com sucesso. Como se pode notar o sistema operacional executa as chamadas de sistema. A figura 2.2.1 pode auxiliar a esclarecer melhor esta situação.

Existem chamadas de sistema destinadas a lançamento de processos, gerenciamento de memória, leitura e escrita de arquivos e controle de entrada/saída de dados.

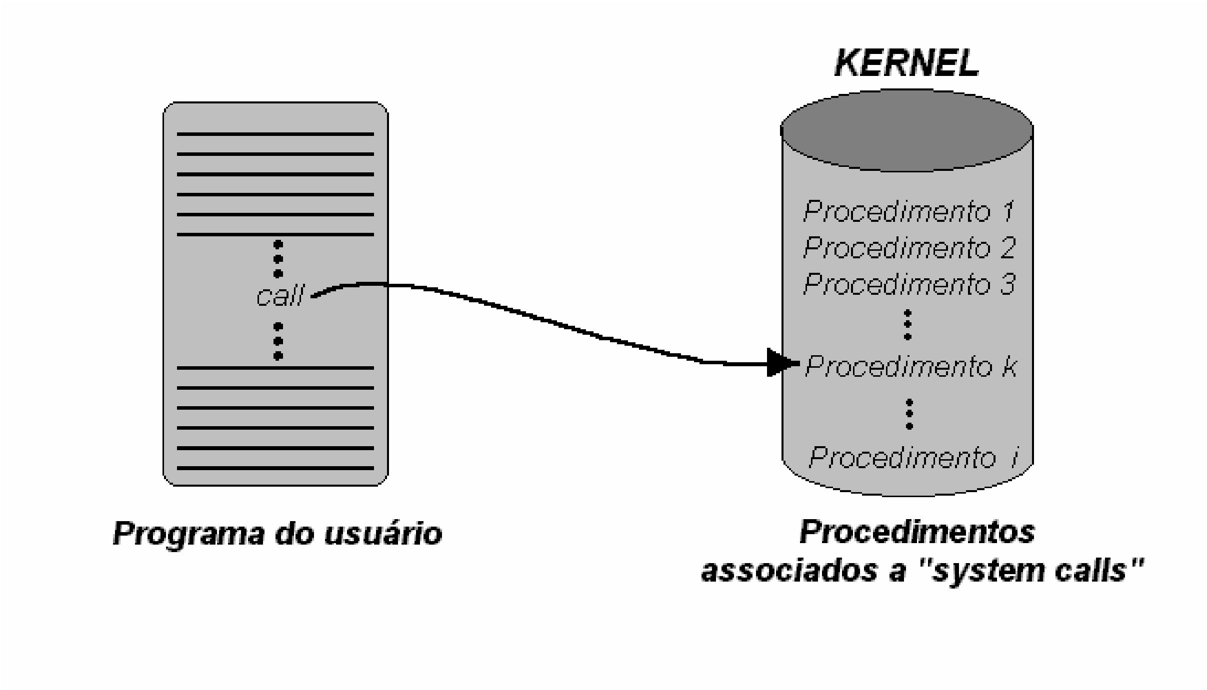


Figura 2.2.1 - Mecanismo das "system calls"

2.3) - PROCESSOS no LINUX

Um conceito principal em todos os sistemas operacionais diz respeito ao **processo**. Processos são simplesmente programas em execução. Um processo é composto de cinco itens a definir :

1. ***Programa executável ;***
2. ***Dados / Pilha ;***
3. ***Contador de programa (PC) ;***
4. ***Apontador de pilha (SP) ;***
5. ***Registradores.***

Estes cinco itens constituem as informações necessárias para a execução do programa. Formam o contexto de execução que caracteriza o processo e que tornam possível interrompê-lo e voltar a executá-lo algum tempo depois, se necessário.

Um exemplo clássico seria o processo que executa o programa do interpretador de comandos (SHELL). Este programa tem a função de ler os comandos de um terminal e criar, por exemplo, outro processo que execute um outro programa qualquer para o atendimento do comando do usuário (poderia ativar o compilador, por exemplo).

Os computadores modernos têm a capacidade de realizar diversas tarefas ao mesmo tempo. Enquanto se executa um programa de algum usuário pode-se estar lendo dados de arquivo localizado no disco e imprimindo-os em uma impressora, por exemplo. Estes processos podem estar executando de maneira concorrente, de forma que a CPU interrompe periodicamente cada um deles para poder atender aos outros presentes naquele instante de tempo. Cada vez que se seleciona um processo para execução é necessário guardar as informações relativas ao estado do processo que está tendo sua execução interrompida para que se possa, futuramente, voltar a executá-lo (esta operação é chamada de "chaveamento entre processos").

Basicamente um processo pode assumir três estados diferentes : executando (ou ativo), pronto para executar e suspenso. Em geral estes estados são dinâmicos dentro do sistema operacional, ou seja, mudam em função do tempo para cada processo. A figura 2.3.1 mostra as transições possíveis de estados de um processo.

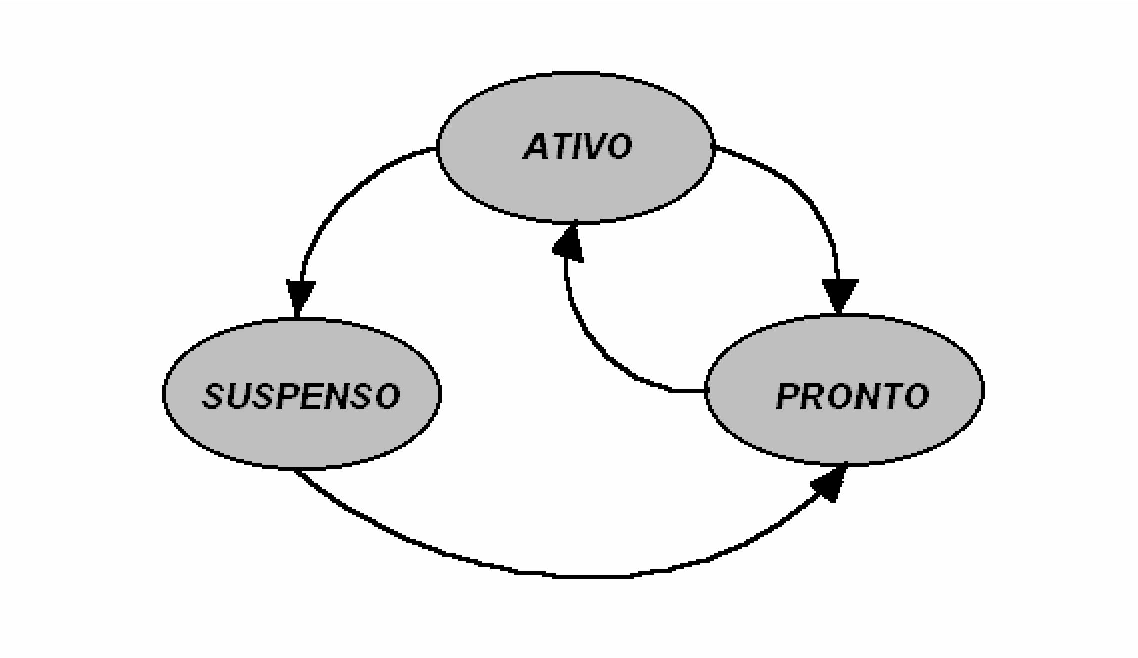


Figura 2.3.1 - Transições de estados nos PROCESSOS

O processo em execução é o centro da atenção da CPU naquele determinado instante de tempo, ou seja, os registradores e as instruções executadas dizem respeito ao processo citado. Os que estão aguardando para serem executados assumem estado de pronto para executar. Isto significa que eles estão em condições de executar porém estão parados temporariamente para que haja o atendimento de outros processos. E, além disso, existem aqueles processos que estão aguardando algum tipo de sinal ou entrada para continuar a sua execução. Estes últimos assumem o estado de suspensos.

Em um sistema multitarefa a CPU faz uma multiplexação entre os vários processos, executando cada um por intervalos de tempo muito curtos. Na verdade, tendo em vista que só existe uma CPU, para cada instante de tempo existe apenas um processo executando. Entretanto, analisando intervalos de tempo maiores, tem-se a impressão de que os processos executam em paralelo. A este fenômeno denomina-se ***pseudo-paralelismo***, contrastando com o paralelismo real que é implementado através de dois ou mais dispositivos de "hardware" operando ao mesmo tempo.

Quem decide a disciplina de atendimento dos processos é o "escalonador de processos" ("scheduler"), que é uma entidade de importância vital para um sistema multitarefa. A política de escalonamento deve atender a diversos requisitos em proporções satisfatórias (a política deve ser justa e eficiente), de modo a garantir a eficiência do sistema. O escalonador se encontra em um nível mais baixo do sistema operacional, e deve ter o menor tamanho possível, para não prejudicar a eficiência dos atendimentos.

O sistema LINUX utiliza um algoritmo de escalonamento em dois níveis. O algoritmo de baixo nível decide qual o processo deve ser executado dentro dos que estão na memória e no estado "pronto para executar". O algoritmo de alto nível move processos entre a memória e o disco, de maneira que todos possam ter a chance de ser executados.

O algoritmo de baixo nível implementa o mecanismo de *múltiplas filas*. Cada fila tem associada um valor de prioridade. Os processos executando no modo usuário possuem valores de prioridade positivo, enquanto os processos executando no modo "kernel" (realizando chamadas de sistema) possuem valores negativos de prioridade. Quanto menor o valor, maior a prioridade associada. Quando o escalonador entra em operação ele procura a primeira fila (com a maior prioridade possível) que esteja ocupada por, pelo menos, um processo. Então os processos desta fila executam por um "quantum" (unidade de tempo dada pela CPU para a execução de um mesmo processo), que tipicamente assume valor de 100 milisegundos, ou até assumir estado de "suspensos".

Neste algoritmo os processos podem pular de classe (há a realocação de prioridades a cada intervalo de tempo definido), em função do tempo de execução. Para cada fila de prioridade utiliza-se o conhecido mecanismo de ***"Round Robin",*** até que se esgote os processos nesta fila (seja porque a execução terminou ou porque os processos pularam de classe de prioridade). Os processos das filas com prioridade menor só passam a executar quando todas as outras filas com prioridades maiores se esgotaram. A figura 2.3.2 mostra um diagrama de como é feita a organização das filas por prioridade. Os resultados conseguidos em sistemas multitarefa com este algoritmo tem se mostrado bastante satisfatórios. Isto também se aplica ao LINUX.

O lançamento de um processo dentro de um sistema operacional pode partir do pedido direto do usuário ao SHELL, pode partir de um programa do usuário ou de algum programa do sistema operacional. Para cada processo lançado temos uma identificação que é estática a partir do início da execução e durante o período em que o mesmo assume um dos três estados descritos anteriormente. Esta identificação é chamada de PID (do inglês "Process IDentification"). O sistema constrói uma tabela contendo a designação de todos os processos correntes em execução ou suspensos. Assim que um processo tem a execução suspensa definitivamente, o seu PID é retirado da tabela de processos. O SHELL do LINUX permite que se tenha acesso a informação de quais processos estão executando e o PID associado a cada um. Para isso digite :

ps <ENTER>

O resultado que será visualizado é uma tabela contendo as informações descritas.

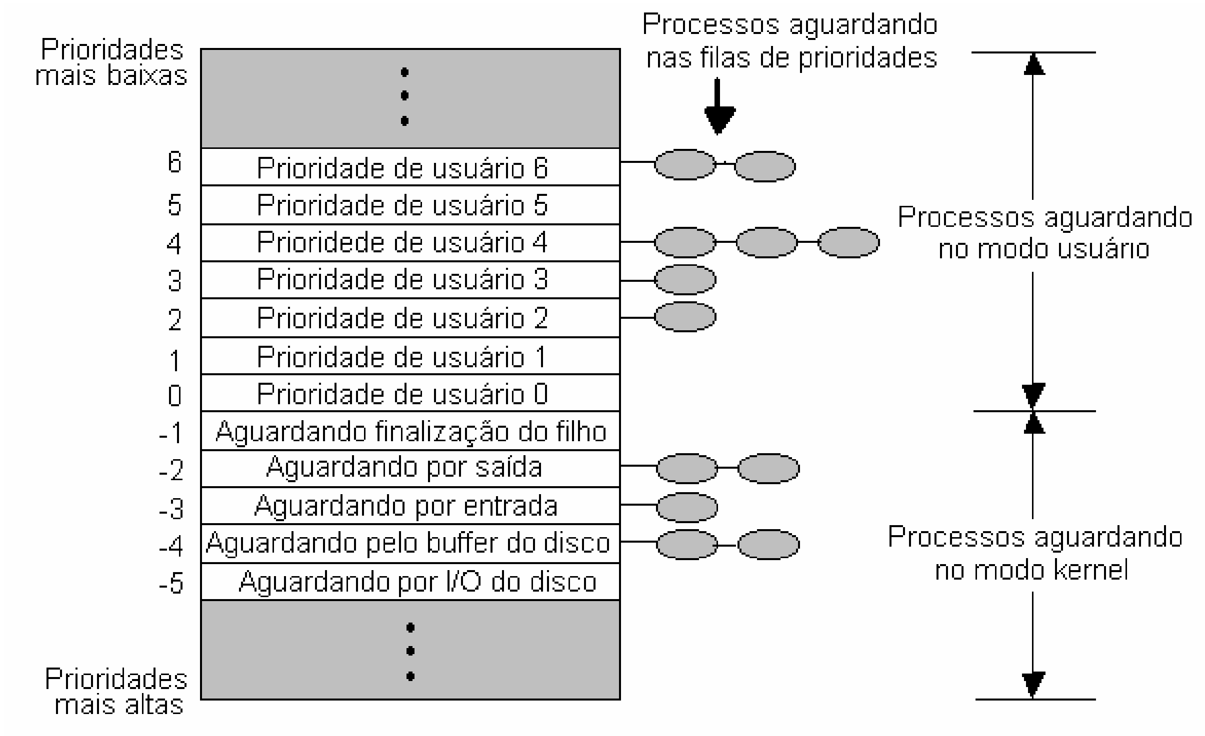


Figura 2.3.2 - Organização dos processos em Múltiplas Filas no LINUX

Os diversos processos no LINUX podem estar executando em “BACKGROUND” ou “FOREGROUND”. Em geral, processos que necessitam ter algum tipo de interação com o usuário (principalmente entrada de dados) são executados em “FOREGROUND”. Por outro lado, processos que não necessitam interação com o usuário podem ser executados em “BACKGROUND” (por exemplo um processo que tenha apenas a tarefa de atualizar algum arquivo no disco). Normalmente apenas um processo lida com a interface mostrada para o usuário, com o intuito de evitar confusão com os ponteiros da tela. O processo em “FOREGROUND” pode ser finalizado através do “CTRL + C”, enquanto os processos que executam em “BACKGROUND” só podem ser finalizados através da primitiva associada (no caso “KILL ( )”). Um usuário, ao solicitar o lançamento de um processo, pode especificar em que nível ele deverá ser executado.

Um outro conceito importante é o de duplicação de um processo. Este tipo de operação tem diversas finalidades. O fato é que, quando se duplica um processo, todas as informações relativas ao mesmo também são clonadas (estas são os cinco itens descritos anteriormente). A partir da duplicação os dois processos (original e cópia ou, na terminologia mais utilizada, PAI e FILHO) passam a executar em pseudo-paralelismo. Os valores das variáveis podem assumir valores diferentes nos dois processos. Cada processo possuirá um PID diferente na tabela de processos.

Uma das aplicações clássicas é a duplicação de um processo que atende conexões em uma rede : cada vez que um novo pedido chega o processo é duplicado e a cópia passa a atender o pedido enquanto o original fica aguardando novos pedidos.

Um outro exemplo interessante seria a resolução de uma equação diferencial com derivadas parciais através da decomposição de domínio. Pode-se duplicar várias vezes um processo e dividir a tarefa de resolução da equação de forma que cada cópia seria encarregada de resolver a equação em uma determinada parte do domínio.

A duplicação de um processo ocorre através da chamada da primitiva específica pelo processo que deseja se duplicar e o sistema operacional cuida de todos os procedimentos necessários para que isso ocorra. Nestes casos o único valor de retorno é o PID do processo cópia recém lançado. Processos cópia também podem se duplicar e assim por diante. Geralmente há um limite no número de processos que podem executar em paralelo a ser respeitado. Este limite varia de um sistema operacional para outro.

Para ilustrar este conceito desenvolvemos o programa “duplica.c”. Em termos gerais, ao lançarmos o respectivo executável através do comando “./duplica” no modo SHELL, este processo se duplicará e a cópia passará a executar em paralelo com o processo original. A função do processo cópia será imprimir na tela três vezes a mensagem “Processo COPIA executando”, enquanto o processo original mostrará a mensagem “Processo ORIGINAL executando”. Como os dois executam virtualmente em paralelo, o que se verá é uma alternância destas duas mensagens na tela. Cabe, neste ponto, fazer algumas considerações acerca de funções e chamadas que serão bastante utilizadas tanto nos exemplos, quanto nos programas do ESTUDO DE CASO. Vejamos :

- Função *“sleep(inteiro t)”*

Esta função, que está definida na biblioteca “unistd.h”, suspende a execução do processo que a evoca durante “t” unidades de tempo a definir. Quando não se explicita o tipo de unidade de tempo, o compilador admite como padrão a unidade de segundos. Por exemplo, vejamos a seguinte chamada :

```
sleep( 2 );
```

Isto fará com que o processo pare por dois segundos, e depois continue a executar.

- Função *“system(char *comando)”*

Esta função, que está definida na biblioteca “stdlib.h”, executa um comando no modo SHELL, como se você estivesse digitando este comando no “prompt”. Para isso a “string” apontada pelo argumento deve conter todas as informações necessárias para a execução do comando. Uma chamada muito utilizada nos exemplos é a seguinte :

system(“clear”);

Isto fará a limpeza da tela, equivalente ao “clrscr” do PASCAL. De fato esta não é a maneira mais elegante de realizar tal tarefa, porém é a mais prática e a menor em termos de linhas de código, o que atende ao propósito dos EXEMPLOS desenvolvidos.

Uma outra chamada utilizada é a seguinte :

system(“ps”);

Este comando mostra a lista dos processos em execução naquele instante, evidenciando o PID de cada um. Como pode ser visto o próprio processo “ps” é parte integrante desta lista, já que está sendo executado quando se mostra a lista com os processos em execução.

- System call *“exit( )”*

Esta chamada, que está definida na biblioteca “unistd.h” faz com que termine a execução do processo que faz a chamada.

Tendo enfatizado isso, vamos comentar as funções que tratam da duplicação de um processo. Vejamos :

- System call *“getpid( )”*

Esta chamada de sistema, que está definida na biblioteca “unistd.h”, retorna um valor do tipo inteiro contendo o PID do processo que fez a chamada. A utilização é bastante simples e da forma (pid é uma variável genérica do tipo inteiro):

pid = getpid( );

- System call *"fork( )"*

Esta chamada de sistema, que está definida na biblioteca "unistd.h", realiza a duplicação do processo que fez a chamada, retornando um valor inteiro contendo o PID do processo em execução. A partir deste momento, uma cópia do processo passa a executar em paralelo. Para que se possa distinguir entre processo original e processo cópia (comumente chamados de PAI e FILHO), verifica-se o valor de retorno da função. Vejamos a seguinte chamada :

pid = fork ( );

A partir de então há a duplicação do processo. Se o processo PAI estiver sendo executado, o valor de "pid" será igual ao PID do processo FILHO. Do contrário, se o FILHO estiver sendo executado, então o valor de "pid" será igual a zero. Como já foi estudado, um processo é composto de vários itens, dentro os quais a região contendo as variáveis utilizadas pelo mesmo. No processo PAI a variável "pid" recebe o valor descrito, enquanto que, no processo FILHO, recebe o valor zero. Caso ocorra algum erro na duplicação do processo teremos "-1" como valor retornado na variável "pid".

Tendo visto estas funções já é possível comentar o programa "duplica.c".
Após definir as variáveis e inicializar a tela, é feita a chamada :

pid_original = getpid( );

Logo após é feita a chamada da função "fork ( )". Então verificamos a variável "pid". Caso seja maior que zero, o que significa que o PAI está executando, mostra-se os PIDs dos dois processos, temos um loop que mostrará três vezes a mensagem "Processo ORIGINAL executando" e, logo depois, veremos a execução do comando "ps". Por outro lado, caso seja igual a zero, o que significa que o FILHO está executando, teremos um loop que mostrará três vezes a mensagem "Processo COPIA executando". As chamadas que realizam estas tarefas estão mostradas abaixo (extraídas diretamente do programa e reproduzidas aqui por conveniência) :

pid = fork( );

if (pid == 0) //copia executando

```
{
  sleep(1);
  for (contador=1 ; contador<=3;contador++ )
  {
     printf("%49s\n", "Processo COPIA executando");
     sleep(2);
  }
   sleep(1);
}
else if (pid>0) // original executando
{
  printf("ORIGINAL tem PID = %d\n", pid_original);
  printf("COPIA        tem PID = %d\n", pid);
  for (contador=1 ; contador<=3;contador++ )
  {
     printf("%49s\n", "Processo ORIGINAL executando");
     sleep(2);
  }
  printf("\nVerifique o PID da COPIA na tabela abaixo :\n");
  system("ps");
}
```

Como fica evidente na execução do "ps" temos dois processos "./duplica" executando em paralelo. Caso ocorra algum erro na duplicação do processo veremos na tela a mensagem "Houve algum erro na duplicação do processo…", situação evidenciada pelo valor de retorno "-1". Este é um exemplo bastante simples, mas que evidencia muito bem o conceito de duplicação de processos.

Vamos apresentar, agora, um programa que realiza a triplicação de um processo.

Este programa se encontra sob o título "triplica.c". Existe uma forte semelhança entre este exemplo e o anterior. Neste caso, haverá o lançamento de duas cópias do processo "./triplica". Para tornar mais fácil a visualização do código foi desenvolvida a função "loop_copia (int num_copia)", que mostra na tela três vezes a mensagem "COPIA (num_copia) executando". Após a primeira chamada do "fork ( )", chamamos a função "loop_copia (1)" ao verificarmos que "pid" é igual a zero, indicando que a primeira cópia estará em execução. De maneira análoga se faz a chamada "loop_copia (2)" após a segunda chamada da função "fork ( )". Vejamos como isto é feito diretamente no código:

```
pid = fork( );

if (pid == 0) // primeira copia executando
{
   loop_copia(1);
    exit(1);
 }
       :
       :
pid = fork( );

if (pid == 0) // segunda copia executando
{
   loop_copia(2);
    exit(1);
 }
```

Pelo que já foi discutido ficou claro que para cada chamada da função "fork ( )" é criada uma nova cópia do processo que faz a chamada. Para evitar que ocorra a duplicação do primeiro processo cópia utilizamos a chamada "exit (1)" quando do retorno da função "loop_copia", garantido que as cópias mostrarão as três mensagens e encerrarão a sua execução. Os artifícios e funções utilizadas neste programa são rigorosamente os mesmos utilizados no exemplo de duplicação. Na verdade este exemplo é um desdobramento do conceito de duplicação de processos. A exemplo do programa anterior pode-se conferir os PID fornecidos a partir da execução da chamada "system ("ps")", presente no final do código.

Da mesma forma que pode ocorrer a duplicação de um processo, pode ocorrer também o lançamento de um outro processo a partir de um em execução. Na verdade, nos sistemas UNIX, a duplicação de algum processo precede o lançamento de um outro. O que acontece é o seguinte : em primeiro lugar o processo se duplica; a seguir as regiões que guardam as informações acerca do processo cópia têm seus dados substituídos pelas informações acerca do novo processo. O PID do novo processo não muda (é o mesmo que era pertencente ao processo cópia). Temos então dois processos distintos executando. O lançamento de outros processos a partir de um em execução é, evidentemente, uma operação de destacada importância e que é comumente utilizada.

Para exemplificar este conceito desenvolvemos dois programas, intitulados "pai.c"e "filho.c".

Aqui, neste exemplo, o que será feito é o lançamento de um outro processo a partir do processo "./pai". Para isso vejamos como utilizar uma função de bastante utilidade.

- System call *"exec( )"*

Esta chamada de sistema, que está definida na biblioteca "unistd.h" transforma o processo que faz a chamada substituindo a sua área de memória com um novo programa. Os argumentos passados se referem a qual programa passará a ser executado e a maneira com que isso será feito.

Conforme podemos notar, temos dois programas distintos. O programa "filho.c" mostra cinco vezes a mensagem "Processo FILHO executando" e termina a sua execução. Execute este programa digitando "./filho" no "prompt" apenas como um teste.

O programa "pai.c", inicialmente, faz a chamada da primitiva "fork ( )". Como já vimos isto causa uma duplicação do processo "./pai". Ao verificarmos que o pid retornado é igual a zero (ou seja, cópia executando), faz-se então a chamada da primitiva "exec( )". Vejamos o trecho do código que faz as chamadas citadas :

```
pid = fork( );

if (pid == 0) //filho executando
{
  printf("Lancando processo ./filho &…\n\n);
  exec ( "./filho", "./filho", "&" , NULL);
   _exit (EXIT_FAILURE); // termina processo se houver algum erro
}
```

Isto causa a substituição da cópia do "./pai" pelo processo "./filho", que passa a executar em paralelo com o processo original "./pai". O terceiro argumento passado ("&") indica que o processo será executado em **"BACKGROUND"**. Na continuação do processo "./pai" temos um loop que exibe cinco vezes a mensagem "Processo PAI executando". Assim como nos dois exemplos anteriores podemos conferir os PIDs mostrados a partir da execução da chamada " system ("ps") ". Agora veremos na tabela os dois processos distintos "./pai"e "./filho &".

Como podemos notar este exemplo é uma generalização do conceito de lançamento de processos. Agora já estamos capacitados a fazer cópias de um processo em execução assim como lançar outros processos a partir do mesmo. Note que diversas outras situações poderiam ser exemplificadas a partir destes exemplos sem maior dificuldade como, por exemplo : duplicação de processos filhos, lançamento de filhos de processos filhos, múltiplas cópias, entre outras.

Os exemplos apresentados até aqui serão intensamente utilizados na solução do nosso problema, a qual se baseará em uma modelagem multitarefa, onde estarão executando processos diferentes e cópias de alguns destes processos em paralelo.

Por fim, notemos que pode haver a necessidade de finalizar processos que foram duplicados ou lançados. Esta também é uma operação bastante comum. Podemos finalizar processos a partir de um processo em execução ou a partir do SHELL.

Uma situação muito comum é o travamento de algum processo em uma WORKSTATION ligada à rede. Se este processo, ao travar, não permite que o usuário o finalize através do "Ctrl + C" (trava a máquina), geralmente é necessário acessar outra máquina, entrar na conta e finalizar o processo defeituoso. Isto fará com que a máquina travada seja liberada.

Outra situação possível se verifica quando um processo do usuário lança outros processos e chega ao fim de sua execução. Imagine que os processos lançados continuem executando em "BACKGROUND" e que isso não seja desejável. Neste caso há meios de se finalizar todos os processos lançados antes de terminar a execução.

Existe uma gama de outras possíveis situações. O fato é que a finalização de processos pode ser uma operação vital dentro de um sistema.

Para ilustrar o conceito de finalização de processos desenvolvemos o programa "finaliza.c".

Este programa serve para demonstrar dois tipos de finalização de processos, através da já comentada "exit( )" e através da chamada de sistema "kill( )". Vejamos como ela pode ser utilizada :

- System call *"kill (pid, SIGKILL)"*

Esta chamada de sistema envia um sinal de finalização para o processo cujo PID está especificado como primeiro argumento. Em geral, quando há a duplicação de um processo, o processo PAI tem a informação do PID da cópia lançada; este valor pode ser utilizado nesta chamada para finalizar a cópia que se lançou. A constante "SIGKILL" está definida na biblioteca "signal.h".

Neste exemplo há a duplicação do processo PAI, sendo que a cópia executa um "loop" infinito sem nenhum comando interno. Vejamos como isto é feito :

```
pid = fork ( );
if  ( pid == 0 ) // copia executando
{
```

```
printf("Processo COPIA em loop infinito...\n\n");
for( ; ; ) // loop infinito
{
}
}
```

Após a duplicação e a apresentação da tabela de processos correntes o processo PAI finaliza o processo cópia através da utilização da chamada descrita anteriormente. A chamada é a seguinte :

```
kill(pid,SIGKILL);
```

Quando se visualiza pela segunda vez a tabela de processos correntes (depois do comando KILL( )) podemos notar que a cópia do processo possui o estado "./finaliza <zombie>". Isto significa que o processo foi suspendido e não executa mais. Apesar disso, tendo em vista que o PAI deste processo ainda está em execução, o sistema informa que a cópia teve execução suspensa (daí a designação "zombie" (zumbi)). Logo após isto o processo se auto-finaliza através da chamada da system call "exit(0)". Este exemplo é bastante simples e auto-explicativo. Execute-o através da digitação de "./finaliza" e acompanhe as mensagens que serão mostradas durante a execução.

2.4) - Comunicação entre processos

A comunicação entre processos é uma situação muito comum em sistemas multitarefa. Ela pode ser feita entre processos que executem em uma mesma máquina ou processos que estejam executando em máquinas diferentes.

O próprio acesso disciplinado à regiões compartilhadas é uma forma de comunicação entre processos, onde a informação básica em trânsito é se existe alguém acessando determinada região no momento em que algum processo tente acessá-la. Esta informação deve estar disponível para todos os processos de modo que, quando alguém inicia o acesso, alguma variável comum é ajustada de maneira a demonstrar esta situação. Esta situação evidencia uma comunicação entre os processos.

Existem várias formas de se implementar comunicação entre processos. Para processos executando localmente pode-se utilizar os "pipes". Um "pipe" é um pseudo-arquivo que serve para conectar dois processos. Estes processos podem escrever e ler informações deste arquivo. Neste caso a comunicação entre os processos se assemelha a uma simples operação de escrita e leitura em um arquivo.

Outra forma de comunicação muito utilizada é a troca de mensagens. Neste caso podemos comunicar dois processos que executam em máquinas diferentes também (basta que estas máquinas estejam conectadas à mesma rede). Para processos locais podemos utilizar as filas de mensagens. Como o próprio nome já diz podemos criar um "buffer" que conecta dois processos e, a partir disto, enviar e receber mensagens. Para isto são utilizadas "system calls" específicas como, por exemplo, "send( )"e "receive( )".

Entretanto, quando desejamos conectar dois processos que executam em máquinas diferentes, o método mais aconselhado é o uso de SOCKETS. Este é um método geral que também pode comunicar processos locais. Embora a implementação de SOCKETS não seja tão trivial, a sua utilização pode ser justificada pela eficiência e facilidade de uso. Os SOCKETS podem implementar comunicação orientada à conexão ou não. Existem vantagens e desvantagens das duas técnicas. Vamos citar algumas delas abaixo :

- *Técnica orientada à conexão :* Maior confiabilidade uma vez que é estabelecida uma conexão; os pacotes chegam de forma ordenada; há menos "overhead" no pacote; tempo de processamento menor; não há flexibilidade de rota, ou seja, se a rota estabelecida inicialmente passa a ser defeituosa o sistema não transmitirá os dados de maneira correta;

- *Técnica não orientada à conexão* : Maior "overhead" nos pacotes; tempo de processamento maior; não possui rota fixa, o que pode ser vantagem no caso de falhas; permite o "broad-cast" em uma rede local;

Uma vez estabelecida uma conexão podemos fazer uso das chamadas "read" e "write", que têm sintaxe bastante simples. Para ilustrar a implementação deste tópico desenvolvemos um exemplo que será apresentado agora.

Sem sombra de dúvida este exemplo é o mais importante e o mais complexo dentre os apresentados neste capítulo. Ele implementa uma arquitetura CLIENTE-SERVIDOR através da comunicação por troca de mensagens entre dois processos utilizando SOCKETS (do inglês "encaixe"). Na verdade escolhemos este método para implementação de comunicação entre processos pelo fato de que é de grande generalidade. Com este método podemos comunicar tanto processos que executam na mesma máquina como processos que executam em máquinas diferentes, o que é essencial para o tipo de solução que adotaremos posteriormente. Os SOCKETS constituem uma interface definida por Berkley para acesso à redes que utilizam TCP/IP entre outros protocolos e implementam transporte de dados com orientação à conexão ou não. O que nos levou a escolher os SOCKETS com orientação à conexão foi o aspecto da confiabilidade. Com o estabelecimento de conexões nos SOCKETS podemos garantir, com maior faixa de segurança, que os dados enviados chegarão ao seu destino com integridade. Neste aspecto, a confiabilidade pode facilitar o tipo de protocolo usado para troca de dados. Não estaremos focando com detalhes, neste exemplo, o tipo de aplicação desenvolvida; o que se pretende é dar ênfase ao modo pelo qual é possível estabelecer uma conexão, o formato utilizado pelas mensagens que trafegam pela rede, o modo de transmissão e recepção e o aspecto do acesso concorrido de vários CLIENTES a um mesmo SERVIDOR.

Conforme já citado estaremos implementando uma arquitetura CLIENTE-SERVIDOR. Para criar um SERVIDOR devemos seguir os seguintes passos :

1. Criar um novo SOCKET através da chamada de sistema : "socket( )";
2. Amarrar um endereço (Endereço IP + número da porta) ao SOCKET criado através da chamada de sistema : "bind". Este passo identifica o SERVIDOR, de modo que o CLIENTE possa saber chegar até ele;
3. Entrar em modo de escuta para novos pedidos de conexão no SOCKET, através da chamada de sistema : "listen( )";
4. Aceitar novas conexões através da chamada de sistema : "accept( )".

Em geral, o atendimento de um CLIENTE após o estabelecimento de uma conexão pode demandar um considerável intervalo de tempo. Por isso seria mais eficiente, neste caso, aceitar e lidar com novas conexões enquanto um pedido está sendo atendido. Neste ponto inserimos o conceito de acesso concorrido a um mesmo SERVIDOR. A técnica mais comum de implementar este tipo de situação é a auto-duplicação do SERVIDOR (lançamento de um processo cópia) para atender a nova conexão estabelecida. Sendo assim, para cada CLIENTE que faz um pedido de conexão temos uma cópia do SERVIDOR para atendê-lo. Teoricamente este tipo de implementação admitiria a conexão de infinitos CLIENTES ao mesmo tempo, não fosse a limitação da memória e o limite de processos lançados que é característica imposta pelo sistema. A implementação do SERVIDOR nos moldes descritos acima é realizada através da função "int open_server_socket (int serverport)", que está definida na biblioteca criada "s_socket.h". O argumento passado para a função é a porta em que as conexões deverão ser estabelecidas. Vamos comentar alguns pontos importantes acerca desta função.

A função inicia criando um novo SOCKET usando a chamada de sistema "socket( )", da seguinte forma :

```
if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
        perror("server : can't open stream socket");
```

Esta chamada passa três argumentos : o domínio no qual se deve escutar, o tipo de SOCKET e o protocolo de rede.

O domínio pode ser tanto o AF_UNIX (isto é, apenas o domínio interno à máquina local) quanto o AF_INET (isto é, todos os pedidos provenientes da INTERNET).

O tipo do SOCKET especifica a semântica de comunicação da conexão. Enquanto poucos tipos de SOCKETS foram especificados, na prática, o SOCK_STREAM e SOCK_DGRAM são as implementações mais comuns. O tipo SOCK_STREAM é utilizado para comunicações orientadas à conexão, enquanto SOCK_DGRAM é utilizado para comunicações não orientadas à conexão.

O parâmetro de protocolo identifica o protocolo particular a ser utilizado com o SOCKET. Embora múltiplos protocolos possam existir dentro de uma dada família de protocolos (ou domínio), geralmente só existe um. Para orientação à conexão temos o IPPROTO_TCP, enquanto por outro lado temos o IPPROTO_UDP. Não é necessário especificar explicitamente este parâmetro ao se fazer a chamada da função. Ao invés disso, a utilização do valor zero selecionará o protocolo padrão.

Uma vez que o SOCKET foi criado pode ser operado por meio das opções do SOCKET. Existe uma série de opções de atribuição a um SOCKET. No nosso exemplo, o SOCKET é ajustado de modo a reusar antigos endereços (isto é, endereço IP + porta) sem a necessidade de esperar o tempo requerido para "timeout" do fechamento da conexão. Este ajuste é feito através da chamada :

setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char *) &on, sizeof(on));

Utilizamos um arquivo separado denominado "hostname.h", o qual contêm o "host name" da máquina em que estamos utilizando o programa.

Então o SERVIDOR constrói um endereço para o SOCKET através da resolução do "host name" em um endereço de domínio INTERNET. O endereço IP do SERVIDOR pode ser, ao invés disto, ajustado para INADDR_ANY para permitir que um CLIENTE possa contatar o SERVIDOR em qualquer dos seus endereços IP. O pedaço do código abaixo realiza esta tarefa :

```
serv_addr.sin_family          = AF_INET;
serv_addr.sin_addr.s_addr  = htonl(INADDR_ANY);
serv_addr.sin_port             = htons(serverport);
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr) )
```

Após a criação de um endereço, ele é atribuído ao SOCKET. O SOCKET pode agora ser utilizado no modo de escuta de novas conexões. O BACK_LOG especifica o tamanho máximo da fila de conexões pendentes. Se um pedido de conexão chega quando a fila está cheia, ele será recusado. O BACK_LOG é o segundo argumento na chamada utilizada no nosso exemplo :

```
listen(sockfd , 5);
```

Estando em modo de escuta o SOCKET se prepara para aceitar os pedidos de conexão e atendê-los. Neste exemplo utilizamos um "loop" infinito (utilizando for(;;)). Uma vez que uma conexão é aceita, o SERVIDOR se auto-duplica, de modo que a cópia continua a execução e atende o CLIENTE, enquanto o processo original continua em modo de escuta para atender outros eventuais pedidos. O trecho do código abaixo mostra este "loop" :

```
for ( ; ; )
{
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  if ( newsockfd<0 )
        perror("server : accept error");
  if ( ( childpid = fork( )) < 0)
        perror("server : fork error");
  else if (childpid == 0)
  {
     close(sockfd);
     return(newsockfd);
  }
   close(newsockfd);
 }
```

A criação de um CLIENTE, no que diz respeito ao estabelecimento de conexão, é relativamente mais simples (até mesmo pelo tamanho do código). Implementamos um CLIENTE através da biblioteca "c_socket.h", a qual possui a função "int open_client_socket (int clientport)". Mais uma vez o argumento passado é a porta a ser utilizada. É evidente que a porta deve ser a mesma utilizada pelo SERVIDOR para que possa ser estabelecida conexão (no nosso exemplo utilizamos a porta 40000). Mais do que isso, é necessário que exista algum SERVIDOR executando antes de que algum CLIENTE faça qualquer pedido de conexão, para que ela possa ser atendida. Assim como no SERVIDOR, um SOCKET é criado. Após localizar o SERVIDOR, a chamada "connect( )" estabelece uma conexão "duplex", ou seja, podemos enviar e receber mensagens através dela. A sua utilização é feita da seguinte forma :

```
if (connect(sockfd,(struct sockaddr *) &serv_name, sizeof(serv_name)) < 0)
        return(0);
return(sockfd);
```

Agora que entendemos como estabelecer uma conexão, vamos comentar algo a respeito da nossa aplicação. O que implementamos é o envio de um conjunto de caracteres (no máximo 90) a ser operado pelo SERVIDOR. As operações possíveis são as seguintes :

- Retornar os caracteres nas posições pares (incluindo a posição zero);
- Retornar os caracteres nas posições ímpares ;
- Retornar a seqüência enviada em ordem inversa.

Para cada função temos um caracter indicativo. O formato de uma mensagem que trafega na rede é o seguinte :

[tamanho da mensagem][caracter indicador de função]#[dados]#

Este formato é proprietário da nossa aplicação. Apesar disso, a técnica de se adicionar o primeiro campo com o tamanho do resto da mensagem é amplamente utilizada.

A transmissão de uma mensagem é feita através da chamada "write( )", na forma :

```
write(sockfd, mensagem, tamanho da mensagem);
```

O primeiro parâmetro passado é o descritor do SOCKET, o segundo é a mensagem propriamente dita, e o terceiro é o tamanho da mensagem a ser transmitida. Um aspecto interessante é que a chamada "write( )" não trava o processo até que o processo destino receba a mensagem; daí a necessidade da implementação de algum protocolo para sincronização dos processos.

Por outro lado a recepção de uma mensagem é feita através da chamada "read( )". Os parâmetros passados são os mesmos da função "write( )". Mas como saber o tamanho da mensagem a ser recebida? Daí a utilização do campo de "tamanho da mensagem". Geralmente reserva-se os dois ou três primeiros bytes da mensagem para conter o tamanho do resto da mensagem. Ao ler estes bytes o processo a receber a mensagem pode passar os parâmetros corretos para a função "read( )". No código do CLIENTE este mecanismo é implementado da seguinte maneira :

```
// recebe mensagem do SERVIDOR
printf("Aguardando resposta do SERVIDOR…\n");
```

```
sleep(3);
read(sockfd, tam_chr, 2); // le tamanho da mensagem da resposta
tam_resposta = atoi(tam_chr);
read (sockfd, mens_servidor, tam_resposta); // le resposta
```

No código do SERVIDOR implementamos este mecanismo de forma análoga. Como pode ser visto reservamos os dois bytes iniciais da mensagem para conter o tamanho medido em número de bytes.

Sendo assim, o restante do exemplo diz respeito ao modo como preparamos a mensagem a ser enviada e a maneira com que realizamos as tarefas requisitadas na mensagem.

Este exemplo é bastante ilustrativo e completo. Para executá-lo, primeiramente, digite "./servidor &" no prompt (execução do SERVIDOR em BACKGROUND). Depois digite "./cliente" e siga as instruções : o programa é auto-explicativo. Se desejar conectar mais de um CLIENTE ao mesmo tempo, abra outras seções e conecte quantos CLIENTES quiser.

2.5) - Acesso à regiões compartilhadas de dados

Como já sabemos a execução de vários processos em pseudo-paralelelismo é perfeitamente possível em alguns sistemas operacionais modernos. Estes processos, em geral, operam sobre dados que podem estar localizados em alguma região de memória ou do disco. Não é situação incomum que mais de um processo faça a leitura ou alteração de dados localizados em um mesmo local. A esta situação denominamos acesso concorrido. Se os dados estiverem em região de memória então esta região é chamada de memória compartilhada. Se os dados estiverem em um arquivo então este é chamado de arquivo comum. A competição por recursos onde o instante de acesso pode determinar o resultado final é comumente chamada de "race condition". A figura 2.5.1 ilustra uma situação onde três processos competem no acesso a uma região compartilhada de dados.

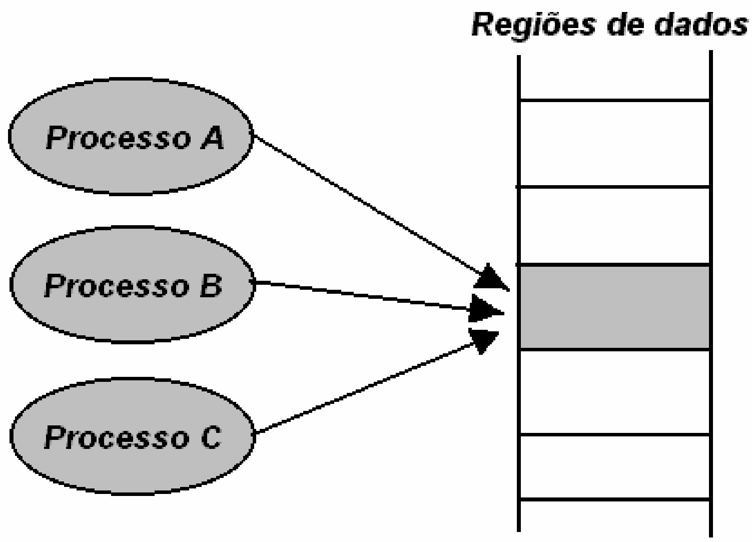


Figura 2.5.1 - Acesso concorrido a região compartilhada de dados

Em cada processo existem partes que realizam o acesso concorrido à regiões comuns. Estas partes do programa são denominadas "regiões críticas". Muitas pessoas confundem este termo associando-o, erroneamente, às regiões onde estão localizados os dados comuns (memória ou arquivo). Para que esta confusão não ocorra mais organizamos as definições abaixo :

- ***Regiões compartilhadas de dados*** : São os lugares onde estão localizados os dados que serão acessados por mais de um processo. Podem ser arquivos compartilhados ou regiões de memória compartilhada.

- ***Regiões críticas*** : São as partes dos programas onde as regiões compartilhadas são acessadas.

O acesso à uma região compartilhada de dados deve ser realizada de maneira organizada. Caso isto não seja feito há a possibilidade de que os dados consultados estejam corrompidos. Afinal, enquanto estamos lendo alguma informação, existe a probabilidade de que algum outro processo esteja alterando estas mesmas informações. No caso de sistemas em tempo real isto se torna ainda mais crítico. Desta forma, o acesso à estas regiões deve ser feito por um processo de cada vez. Enquanto isto ocorre os outros processos que desejam acessar a região compartilhada devem esperar até que o acesso do primeiro termine (e a região seja liberada). A este método dá-se o nome de "acesso disciplinado com exclusão mútua". Em muitos casos a disciplina de acesso obedece à filosofia "chega primeiro, acessa primeiro" (FIFO).

Existem muitos algoritmos que foram desenvolvidos com o objetivo de garantir acesso disciplinado à regiões compartilhadas de dados. Todos eles têm suas vantagens e/ou desvantagens em relação aos outros. Algumas soluções são de mais baixo nível, enquanto outras utilizam níveis de abstração mais altos. De uma maneira geral, o acesso disciplinado tem embutido algum tipo de comunicação entre os processos envolvidos.

Uma forma de comunicação entre processos que pode garantir exclusão mútua para acesso disciplinado é realizada através dos "semáforos". Esta solução foi sugerida por E.W.Dijkstra (1965). A idéia básica era a de utilizar uma variável inteira não negativa para controlar o acesso de vários processos que estariam na fila de espera a alguma região compartilhada. Esta variável deveria ser de um tipo especial, já que é de domínio público dos processos. Para se alterar o valor desta variável foram criadas algumas chamadas de sistema específicas. Vamos descrever, a seguir, as três principais (neste caso, "s" é uma variável do tipo semáforo):

- ***DOWN(s)* :** Se (s>0) então "s" é decrementado de um e a execução do processo continua;
  Se (s=0) o processo passa do estado "ativo" para o estado "suspenso" e "s" não é alterado.
- ***UP(s)* :** Se há processos suspensos no semáforo "s", um desses processos passa para "ativo" e o semáforo "s" não é alterado;
  Se não há processos suspensos então "s" é incrementado de um;

- ***INITSEM(s)*** : Inicializa o semáforo com algum valor.

Note que, no formato mostrado abaixo, poderemos ter exclusão mútua para acesso à região compartilhada (desde que, inicialmente, o semáforo assuma valor igual a um) :

...
down(s) ;
Código contendo a Região Crítica do processo;
up(s) ;
...

Esta técnica dos semáforos é utilizada com razoável freqüência. Entretanto ela só é válida para processos que executem em uma mesma máquina. Vamos apresentar um programa que exemplifica a implementação deste tópico.

A utilização de semáforos como forma de organizar o acesso à áreas compartilhadas de dados é um método bastante eficiente. Entretanto a sua implementação não é tão trivial como se poderia imaginar. Na verdade há uma seqüência que deve ser seguida para que se possa utilizar esta ferramenta. O nosso objetivo principal ao elaborar este exemplo foi o de fornecer as rotinas que implementam esta facilidade permitindo a sua utilização. Entrar no mérito de todas as variações e operações possíveis sobre este tema demandaria tempo e complicações desnecessárias, fugindo à proposta do exemplo. Por isso pretendemos fornecer uma visão geral acerca deste tópico, a qual pode ser aprofundada consultando-se o item **"semget"** do manual do sistema LINUX.

Em primeiro lugar devemos criar um conjunto de semáforos relacionados a algum valor de argumento. Diferente do que se estuda na teoria, devemos nos referir a um conjunto de semáforos através de um valor inteiro, e não através de um nome. Dentro deste conjunto de semáforos podemos realizar operações individuais em cada um deles. Cada semáforo dentro de um conjunto pode ser referenciado através de um outro valor inteiro positivo, sendo que o valor zero se refere ao primeiro semáforo do conjunto. Para exemplificar isso criamos a função "initsem (semkey)" que realiza as seguintes operações : tenta criar um conjunto de um semáforo apenas relacionado ao argumento "semkey". Caso este conjunto ainda não tenha sido criado, então a função inicializa o semáforo com o valor um após criá-lo. O valor retornado é um inteiro ("semid") que será utilizado toda vez que quisermos nos referir a este conjunto de semáforos. Nesta função utilizamos duas chamadas principais:

semid = semget(semkey , 1, SEMPERM|IPC_CREAT|IPC_EXECL) ;

status = semctl(semid , 0 , SETVAL , 1);

A primeira tenta criar o conjunto referido. O primeiro argumento é o utilizado como referência para a criação do mesmo. O segundo argumento indica quantos semáforos este conjunto conterá. O terceiro argumento determina o "status" de permissão de acesso (assim como em um arquivo).

A segunda chamada inicializa o semáforo criado. O primeiro argumento é o valor inteiro a ser utilizado quando nos referirmos ao conjunto de semáforos criado na primeira chamada. O segundo argumento indica qual o semáforo ao qual estamos nos referindo dentro do conjunto criado (como já foi dito o valor zero indica o primeiro semáforo). O terceiro argumento indica que estamos atribuindo um valor ao semáforo referido e o quarto argumento explicita que valor é este. Esta chamada é equivalente ao "INITSEM" descrito anteriormente

O passo seguinte será a implementação das primitivas DOWN( ) e UP( ). Estas primitivas estão implementadas através das funções "down (semid)" e "up (semid)" presentes no nosso exemplo. O argumento passado é, novamente, o valor inteiro utilizado para nos referirmos ao conjunto de semáforos criados com o argumento "semkey". Nas duas funções utilizamos a chamada :

semop(semid, &buf , 1);

O primeiro argumento é o já exaustivamente comentado “semkey”. O segundo argumento é uma estrutura que determina o tipo de operação a ser realizada sobre um determinado semáforo dentro do conjunto ao qual estamos nos referindo. As funções realizam exatamente os procedimentos descritos anteriormente.

Tendo em mãos a função que cria e inicializa o semáforo, e as funções DOWN( ) e UP( ), já é possível exemplificar o uso de semáforos. Para isso criamos uma quarta função chamada “acesso_disciplinado” que simula um acesso concorrido a algum tipo de região compartilhada de dados. É feita a chamada “down (semid)” para garantir acesso exclusivo. Utilizamos a chamada “sleep (10)” para simular o acesso à região comum (na verdade este comando apenas suspende a execução por dez segundos) e logo após liberamos o semáforo através da chamada “up(semid)”.A forma do código escrito é a seguinte :

down(id_sem);
sleep(10);
up(id_sem);

Entretanto o acesso de um único processo a alguma região comum não evidenciaria acesso concorrido. Por este fato voltamos a utilizar o conceito de duplicação de processos. O que se faz é o lançamento de três cópias do processo PAI, os quais deverão simular acesso disciplinado através da função “acesso_disciplinado” comentada acima. Após a execução desta função o processo cópia termina sua execução (através da chamada **exit(0)**). As mensagens que são mostradas na tela durante a execução deste exemplo são auto-explicativas, não necessitando maiores comentários.

2.6) - Comentários

Neste capítulo foram abordados, de maneira sucinta, alguns conceitos teóricos relativos a sistemas operacionais. Abordamos o conceito de “system call”, de processo, de operações com processos, de acesso à regiões compartilhadas de dados e de comunicação entre processos. Estes conceitos serão intensamente utilizados na solução do nosso problema hipotético. Foram apresentados, também, os programas da categoria EXEMPLOS. Eles implementam, de maneira prática, e no ambiente particular do LINUX, os conceitos abordados e que serão utilizados na solução do nosso problema de controle de estoque em tempo real.

Em particular, dentre os programas apresentados, os que serão utilizados com maior freqüência são os que implementam a comunicação entre processos utilizando SOCKETS. Trechos destes programas aparecerão em boa parte dos programas da categoria ESTUDO DE CASO. O exemplo que implementa acesso disciplinado com exclusão mútua utilizando SEMÁFOROS também será muito utilizado na nossa solução. A compreensão dos exemplos apresentados neste capítulo, de uma maneira geral, tornará a análise dos programas do ESTUDO DE CASO muito mais simples.

Agora que já conhecemos estas implementações devemos modelar o nosso problema e propor uma solução que envolva estes conceitos. Isto será feito no próximo capítulo. Esta modelagem é um tanto quanto extensa, assim como interessante e de grande importância para o entendimento do código que implementa nossa solução. Vamos utilizar ferramentas de modelagem bastante úteis, que permitem a validação dos modelos. O próximo capítulo é, em base, uma análise teórica do problema, já tendo em vista a implementação prática da solução proposta.

Capítulo 3 - Estudo de Caso : Modelagem

3.1) - Introdução

O estudo de caso deste projeto é o problema da automação do controle de estoque de um supermercado em tempo real. A modelagem deste problema é um passo essencial para o sucesso de sua resolução. Quando existe um modo de se validar o modelo através de alguma ferramenta de análise a modelagem se torna, particularmente, mais interessante e compensadora. Neste capítulo vamos seguir os passos para uma boa modelagem, partindo da descrição detalhada do problema, caracterizando a solução em termos das ferramentas disponíveis (conceitos implementados no capítulo anterior) e, por fim, validando esta solução modelada. Baseando-se em uma modelagem bem feita, a implementação prática pode ter maiores chances de sucesso e, principalmente, tornar as correções, manutenção e os ajustes mais fáceis de serem realizados tanto pelo próprio programador, quanto por outras pessoas que eventualmente tenham que realizar tais tarefas. É a isto que se presta este capítulo.

Na seção 3.2 procederemos em uma descrição do problema e da estrutura física necessária para a solução. Esta descrição e estrutura física, apesar de hipotéticas, se enquadram perfeitamente dentro de um contexto real.

Na seção 3.3 será proposta, então, uma solução baseada no conceito de processos em um sistema operacional multitarefa. O objetivo será descrever quantos processos deverão ser executados em cada terminal e que tarefas serão realizadas por cada um destes processos.

Na seção 3.4 será realizada uma modelagem destes processos através das REDES DE PETRI. Para que esta possa ser compreendida definiremos, em linhas gerais, as REDES DE PETRI e a relação entre suas propriedades e a validação de sistemas concorrentes. Apresentaremos, então, as redes desenvolvidas para a modelagem do nosso problema e as validaremos utilizando o “software” ARP, desenvolvido especificamente para este propósito.

3.2) - Descrição do problema e da estrutura física disponível

Começaremos nossa modelagem descrevendo um contexto hipotético perfeitamente possível de se enquadrar em uma situação real. Suponha que a gerência de um supermercado resolve acatar os resultados de uma consultoria contratada que, sob o argumento de que isto aumentaria a eficiência e poderia elevar os lucros do estabelecimento, propõe a substituição das antigas máquinas registradoras por terminais PC ligados em rede interna. Deste modo, a leitura dos preços não seria mais tarefa do funcionário do caixa, mas sim de um equipamento de leitura ótica que teria acesso a uma lista contendo a descrição e o preço de cada produto vendido pelo supermercado. Isto poderia diminuir a incidência de erros de digitação de preços, o que é um argumento razoável. Entretanto, suponhamos que além desta facilidade, estivéssemos querendo implementar uma inovação : controle de estoque interno e de produtos no depósito em tempo real. Isto seria feito de modo que ao fim do atendimento de um cliente genérico fosse efetuada baixa no estoque interno de produtos disponíveis nas gôndolas, computando os produtos que tivessem sido comprados pelo cliente. Caso fosse evidenciado que a quantidade estivesse abaixo de um limite mínimo, haveria um pedido automático de reposição que seria enviado para o depósito especificando qual o produto e qual a quantidade a ser reposta nas gôndolas. Deveria haver, neste caso, um monitoramento periódico para se saber quando se deveria fazer um pedido de produtos a um determinado fornecedor externo para reposição no depósito.

Suponha que este sistema tivesse que ter característica de perfeita escalabilidade, de forma que pudessem ser conectados um número bastante grande de terminais caixa sem a necessidade de mudança no “software” desenvolvido. Da mesma forma o número de produtos cadastrados deveria estar situado em um intervalo bastante largo, permitindo que o supermercado ampliasse suas instalações (em alguma ocasião futura) sem a necessidade de fazer qualquer ajuste no “software” utilizado. Além disso, suponha que o sistema pudesse permitir a atualização de preços de produtos durante o expediente, situação presente nas conhecidas “promoções relâmpago”, que são uma tática bastante utilizada para atrair clientes. Nestas situações o gerente autoriza, por um determinado período de tempo (geralmente bastante curto), a redução no preço de um ou mais produtos. Os clientes que passarem nos caixas durante este tempo pagarão o preço promoção abaixo do normal. Estas seriam, basicamente, as exigências feitas para a implementação do projeto.

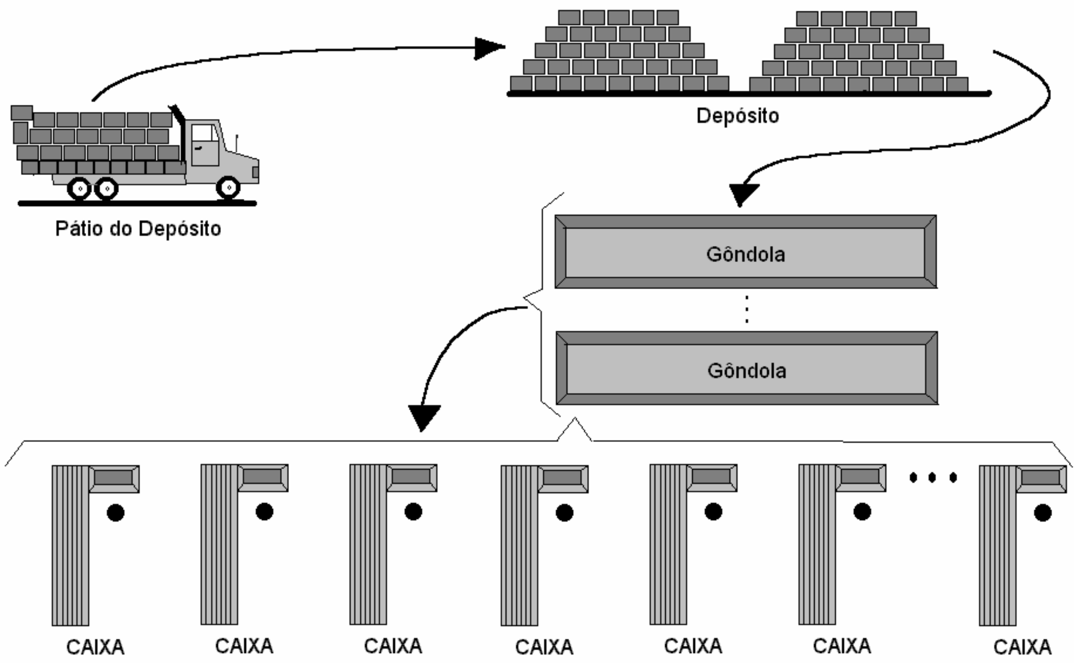


Figura 3.2.1 - Fluxo de produtos

Na fase de descrição seria feito um levantamento de toda a estrutura das instalações, modo de organização interna, número de produtos oferecidos, entre outros aspectos. Vamos descrever uma das configurações possíveis com a qual estaríamos lidando.

Neste supermercado hipotético teríamos um fluxo de produtos que se daria em função das instalações disponíveis : pátio do depósito, depósito para armazenamento, gôndolas e caixas registradoras. A figura 3.2.1 demonstra este fluxo, com as setas indicando o sentido pelo qual os diversos produtos "fluem".

O que pode influir no projeto em termos de organização na forma de dispor os produtos diz respeito ao ambiente interno, ou seja, as gôndolas. Seria aconselhável que todos os produtos iguais fossem dispostos em um mesmo espaço físico reservado, muito embora este espaço pudesse estar localizado em qualquer ponto do estabelecimento. Isto porque seria impossível saber de qual gôndola teria sido retirado determinado produto que fosse registrado em algum caixa (caso o mesmo produto fosse disposto em gôndolas diferentes). Deste modo suponhamos que este supermercado disponha os produtos de uma mesma marca em um mesmo espaço físico.

Cada produto comercializado possui um código de barra único. O que deve fazer é associar a cada código de produto uma descrição e um preço. Estas são as informações que devem estar disponíveis para que um terminal caixa possa operar.

Nestes terminais deveremos ter o sistema operacional LINUX e toda a infra-estrutura em termos de “hardware” já instalada e funcionando, de modo que a nossa solução seria apenas implementada a nível de “software”.

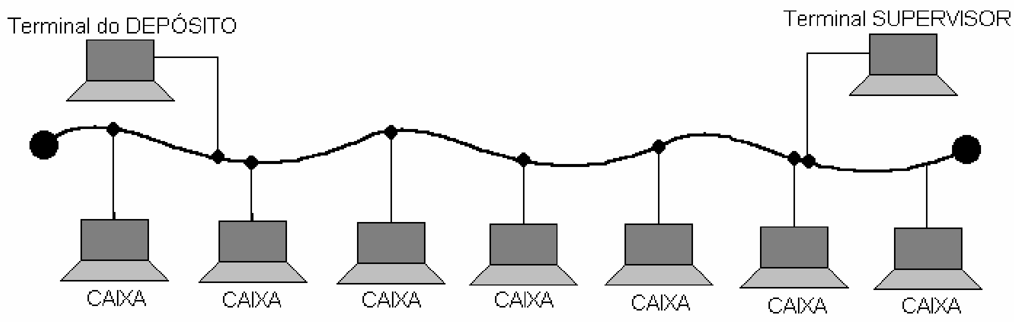


Figura 3.2.2 - Organização dos terminais conectados à rede

Acima podemos visualizar a figura 3.2.2 que mostra como seriam ligados os terminais e aonde eles estariam localizados. Como se vê é sugerida uma rede interna em forma de barramento (ETHERNET, por exemplo) com um terminal localizado no depósito (Terminal do Depósito), um terminal supervisor localizado na sala da gerência (Terminal Supervisor) e diversos terminais caixa localizados como de costume dentro de um supermercado (Terminais CAIXA). Desta forma seria possível estabelecer comunicação entre dois terminais genéricos quaisquer, muito embora a nossa solução estabeleça tipos de conexão bem definidas entre determinados terminais.

Estas seriam as exigências e a estrutura presentes na análise detalhada do problema. Cabe agora propor uma solução utilizando os conceitos estudados relacionados ao LINUX, o que será feito nas páginas seguintes.

3.3) - Modelagem do problema através do conceito de PROCESSOS

Uma das maneiras de se resolver o problema descrito repousa sobre o conceito de PROCESSOS em um sistema multitarefa (como sabemos o LINUX possui tal característica). O que se deve fazer é propor uma solução em função das exigências requeridas. Vamos começar definindo quais as tarefas que serão executadas pelos tipos de terminais que foram descritos no item anterior.

* **TERMINAL do DEPÓSITO** : Este terminal terá apenas a função de receber pedidos de reposição provenientes do terminal SUPERVISOR e autorizar o envio, para o interior do supermercado (gôndola), do produto na quantidade que foi solicitada. Esta autorização pode ser mostrada em forma de uma mensagem impressa na tela seguida de algum aviso sonoro. Deve-se ter um funcionário atento aos pedidos de reposição mostrados por este terminal. Não haverá necessidade de que o funcionário tenha que interagir com o terminal.

* **TERMINAL CAIXA** : Este terminal terá a função de atender a clientes genéricos, registrando os produtos comprados, computando os subtotais e o valor total da compra. Após liberar o cliente este terminal deve enviar para o terminal SUPERVISOR a lista de produtos que foram comprados com as respectivas quantidades. Além disso o terminal CAIXA deve estar atento para as atualizações que são feitas na lista de preços e que são enviadas pelo terminal SUPERVISOR. A atualização da lista só pode ser efetuada nos intervalos em que não estiver ocorrendo o atendimento a um cliente, de modo que o terminal deve checar periodicamente se há atualizações a se fazer. Um terminal CAIXA pode se conectar a uma rede em qualquer momento, assim como se desconectar.

* **TERMINAL SUPERVISOR** : Este terminal é a alma de todo o sistema. Para que os outros possam operar é necessário que este terminal já tenha sido colocado em operação antes. A sua função é a de enviar a tabela de preços para os CAIXAS que se conectam à rede, permitir a atualização de atributos relativos a um ou mais produtos, enviar as atualizações eventualmente realizadas para todos os CAIXAS conectados, receber as listas contendo os produtos que foram comprados pelos clientes, atualizar o banco de dados que contêm as informações sobre cada produto, autorizar reposição interna enviando o pedido para o terminal do DEPÓSITO e permitir que algum funcionário autorizado tenha acesso a todas as informações relativas ao controle de estoque e de preços do sistema. Como se vê, este terminal é o que vai abrigar a maior percentagem dos programas desenvolvidos.

O que devemos fazer agora é definir quantos processos serão necessários para que cada terminal possa desempenhar estas tarefas com boa eficiência. Para cada terminal vamos propor um número determinado de processos e especificar quais seriam as tarefas desempenhadas por cada um destes processos. Para isso podemos visualizar a figura 3.3.1 que mostra os processos e os principais arquivos presentes em cada terminal.

No terminal do DEPÓSITO propomos a execução de um único processo chamado processo para aviso de reposição (PAR). Este processo, então, desempenharia todas as tarefas descritas para este terminal. Este processo é relativamente simples, na medida em que recebe mensagens e as imprime na tela em um formato que possa ser compreendido pelo respectivo funcionário.

No terminal CAIXA propomos a execução de dois processos. Vamos chamá-los de processo CAIXA e processo para tratamento de atualizações (PTA). O PTA, que estará executando em "BACKGROUND", seria encarregado de solicitar o envio da tabela de preços quando da conexão do CAIXA, de receber a tabela e salvar em um arquivo em disco, de enviar o registro do CAIXA e de entrar em modo de escuta, de modo a aguardar as atualizações enviadas pelo terminal SUPERVISOR. Quando houver alguma atualização a se fazer o PTA altera o arquivo em disco e avisa o processo CAIXA de que alterações foram feitas.

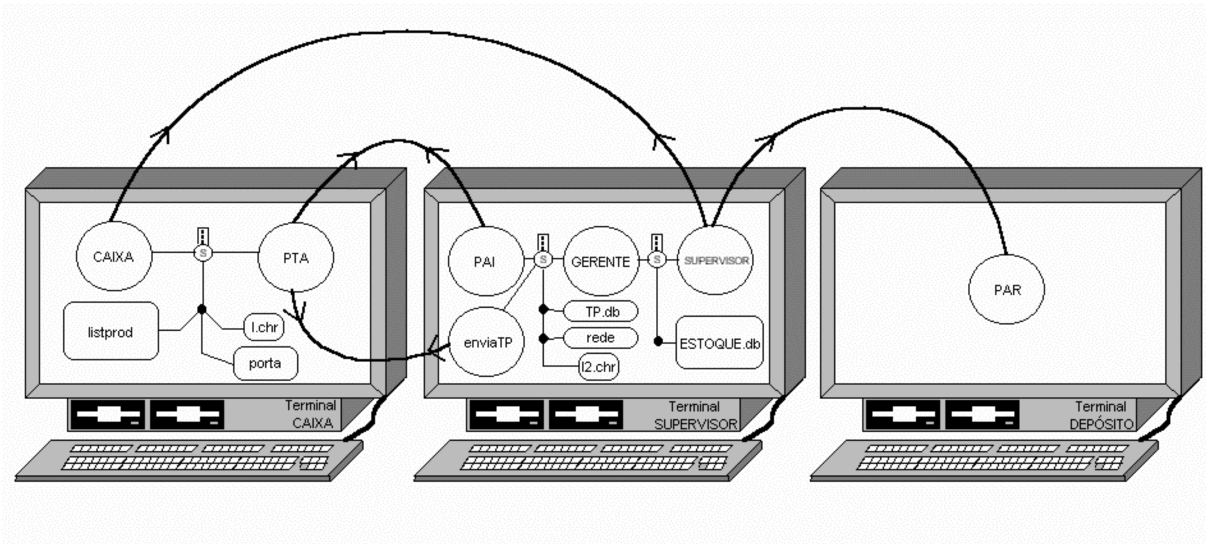


FIGURA 3.3.1 - Processos e Arquivos nos Terminais

O processo CAIXA, que estará executando em "FOREGROUND", deve lançar o processo PTA e aguardar o sinal para carregar a tabela de preços. A partir daí deve cuidar do atendimento dos clientes e verificar, periodicamente, se há a necessidade de recarregar a tabela de preços atualizada do disco. Ao fim da execução do CAIXA deve haver a finalização do processo PTA.

No terminal SUPERVISOR propomos a execução de quatro processos. Vamos chamá-los de : processo SUPERVISOR, processo enviador de tabela de preços (enviaTP), processo de atualização de itens (PAI) e processo GERENTE.

O processo SUPERVISOR, executando em "BACKGROUND", terá função de atender às conexões solicitadas pelos processos CAIXAS presentes nos terminais CAIXA conectados à rede. Ele receberá as listas contendo a informação dos produtos que estão sendo comprados e deverá atualizar o banco de dados com acesso disciplinado. É este processo que envia o pedido de reposição para o PAR do terminal do DEPÓSITO.

O processo enviaTP, executando em "BACKGROUND", fica com a tarefa de enviar a tabela de preços toda vez que um PTA requisitar. O acesso ao arquivo contendo a tabela também deve ser feito de forma disciplinada. Além disso este processo deve registrar o CAIXA que envia o pedido em um arquivo de configuração da rede. Este arquivo será utilizado pelo PAI.

O processo PAI, executando em "BACKGROUND", fica com a função de enviar as atualizações realizadas na lista de preços para todos os PTAs que estão executando nos terminais CAIXA conectados à rede. Para isso faz uso do arquivo de configuração da rede, que possui a informação de quais as portas de comunicação que devem ser utilizadas para a transferência das atualizações.

O processo GERENTE, que será o único executando em "FOREGROUND", será uma típica aplicação de banco de dados com acesso disciplinado aos arquivos de interesse. Aqui neste ponto vamos discutir um pouco acerca do aspecto da estrutura do banco de dados e dos arquivos utilizados no terminal SUPERVISOR.

Conforme pode ser visto no diagrama anterior teremos quatro arquivos que serão acessados no terminal SUPERVISOR pelos diversos processos de maneira disciplinada através de dois semáforos distintos. Vejamos o que conterá cada arquivo :

- ESTOQUE.db : Guardará as informações dos atributos de controle dos produtos. Em cada registro teremos o seguinte formato :

[código do produto]#[quantidade atual na gôndola]#[quantidade mínima na gôndola]# [quantidade atual no depósito]#[quantidade mínima no depósito]#[aviso de falta no depósito]#

Todos estes campos totalizarão trinta e seis bytes. A chave de busca será o código do produto, com cinco números (permitindo, teoricamente, o cadastramento de cem mil produtos).

- TP.db (Tabela de Preços) : Guardará as informações dos atributos básicos de um item. Em cada registro deste arquivo teremos o seguinte formato :

[código do produto]#[descrição do produto]#[preço do produto]#["status" de atualização]#

Estes campos totalizarão cinqüenta e dois bytes reservados. Novamente a chave de busca será o código do produto, assim como no arquivo ESTOQUE.db. O último campo do registro será utilizado pelo PAI para saber se o registro foi modificado ou não. O processo GERENTE, o enviaTP e o PAI concorrerão para o acesso a este arquivo através de um semáforo.

- I2.chr : Arquivo simples de apenas dois bytes, que conterá o indicador “S” ou “N”. Este arquivo servirá de aviso ao PAI, de modo que se for lido o estado “S” significará que existem atualizações a serem enviadas para os PTAs no arquivo TP.db.

- rede : Guardará as informações de registro dos CAIXAS em operação. Em cada registro teremos o seguinte formato :

[número de identificação do CAIXA][porta de comunicação para envio de atualizações]

O processo GERENTE deve lançar todos os outros citados. A aplicação deve permitir a alteração de qualquer atributo de um ou mais produtos, a consulta a informações de um ou mais produtos e sinalizar para o PAI através da alteração do arquivo “I2.chr” quando desejar que as atualizações realizadas sejam efetivamente transmitidas para todos os terminais CAIXA em funcionamento.

Desta forma modelamos a solução através do conceito de processos em um sistema multitarefa e através de acesso disciplinado a regiões comuns. O que se deve fazer agora é modelar cada processo e sua interação com os demais, incluindo toda a parte que tange às trocas de mensagens entre eles. Para isso utilizaremos uma ferramenta estudada no curso e de grande utilidade : as REDES DE PETRI.

3.4) - Modelagem de processos através de REDES de PETRI

O assunto “REDES DE PETRI” é exaustivamente estudado e utilizado para a modelagem na área de programação concorrente e de redes. O que pretendemos, com a nossa compacta explanação teórica, é apenas situar o leitor acerca deste tópico. Ao invés disto poderíamos mostrar logo as redes desenvolvidas para a modelagem da nossa solução, mas acreditamos que isto não seria didático. Deste modo aconselhamos uma consulta às referências fornecidas no fim da documentação para um melhor entendimento deste capítulo.

Em termos gerais, as REDES DE PETRI são uma generalização da técnica de diagrama de estados. O objetivo é mostrar o fluxo de um sistema, evidenciando por quais estados ele passa e quais são os eventos associados. Basicamente, em uma REDE DE PETRI, podemos identificar quatro entidades principais a definir :

- ***Lugares*** : Representados por círculos indicam estados parciais de um subsistema. Estes lugares podem conter uma ou mais fichas e devem estar ligados, pelo menos, a uma transição.

- ***Transições*** : São representadas por pequenos segmentos de retas e indicam mudança de estado relacionado com algum evento ou ação bem definidos. Pode existir várias entradas e/ou saídas em uma mesma transição. Para que uma transição possa ocorrer (na terminologia mais usada "disparada") é necessário que ela esteja "sensibilizada", ou seja, devem haver fichas suficientes em todos os lugares de entrada desta transição.

- ***Arcos*** : São representados por flechas que ligam transições a lugares. O sentido das flechas indica o sentido com que as fichas "fluem". Em algumas implementações são utilizados números para indicar que após determinada transição ter sido disparada são retiradas ou colocadas em determinado lugar uma certa quantidade de fichas, indicada por estes números.

- ***Fichas*** : Abstração que serve para indicar, por exemplo, que o sistema está em um estado parcial ou para representar alguma quantidade disponível.

Sendo assim, uma REDE DE PETRI é um conjunto de vários dos componentes acima e que descreve o funcionamento de um ou mais subsistemas que estão conectados por algum tipo de lugar ou informação comum. Geralmente são dados nomes (associativos) para as transições e para os lugares de modo a identificar as ações e os estados parciais em função das tarefas desempenhadas pelo sistema. Como um sistema pode ser representado por vários processos, uma REDE DE PETRI pode agrupar mais do que um processo. Neste caso dizemos que cada processo é um subsistema. Geralmente uma REDE DE PETRI não assume forma muito complexa, pois não entra no mérito de como as ações acontecem e quais as variáveis envolvidas em algum estado parcial. Ela toma apenas uma amostragem procedual e abstrata.

Relacionado com as REDES DE PETRI temos certas propriedades que podem ser úteis na validação de sistemas concorrentes. Estas propriedades são divididas em duas classes a definir :

Propriedades Gerais : Rede Limitada, Rede Viva, Rede Reiniciável, Rede Conservativa
Propriedades Específicas : Invariantes de Lugar e Invariantes de Transição

Vamos relacionar estas propriedades com as características das REDES DE PETRI que as atendem.

Propriedades Gerais (relativas a uma dada marcação inicial Mo) :

Rede Limitada : Indica que não há acúmulo infinito de fichas em nenhum lugar da rede. Acúmulos infinitos de fichas podem provocar um “overload” nos sistemas, comportamento que não é desejável.

Rede Viva : Indica que há bloqueio na rede. A partir de qualquer marcação há uma seqüência de disparos que dispara qualquer transição.

Rede Conservativa : Indica que o número de fichas presentes na rede se mantém constante. Neste caso não há pontos em que fichas nascem ou morrem.

Rede Reiniciável : Indica que a rede sempre volta ao estado inicial. Isto é importante pois garante que o sistema é cíclico sem bloqueios.

Propriedades Específicas :

Invariantes de Lugar : Indica que é possível estabelecer uma relação linear entre os número de fichas contido em um subconjunto de lugares da rede.

Invariantes de Transição : Indica que é possível encontrar uma seqüência cíclica de disparos de transições.

Em geral convenciona-se que uma REDE DE PETRI deve atender às propriedades gerais para que se possa validá-la de maneira eficiente. A única propriedade em que podemos ser pouco mais flexíveis é a de “Rede Conservativa”. Mesmo que a rede não atenda a esta propriedade não há perigo de que isto venha a causar “dead-locks” ou “live-locks” na rede, impossibilitando a sua validação.

Agora que já estamos familiarizados com a ferramenta de modelagem, podemos apresentar as REDES de PETRI desenvolvidas para modelar a nossa solução através da implementação dos processos descritos anteriormente.

Se notarmos bem, podemos perceber que um pedaço do sistema cuida do controle de estoque, enquanto outro cuida da atualização e transferência da tabela de preços (não por acaso). Para ser mais específico os processos SUPERVISOR, CAIXA e PAR tratam do controle de estoque em tempo real. Por outro lado os processos PTA, enviaTP e PAI tratam da transferência e atualização da tabela de preços. Deste modo resolvemos dividir o sistema em duas redes, cada qual abrigando um dos conjuntos descritos acima. O ponto de interação entre os dois conjuntos são os semáforos utilizados para o acesso concorrido e os arquivos que servem como sinal para processos (I.chr e

I2.chr). O ideal seria construir uma única rede agrupando todos os processos e suas interações. Entretanto a própria elaboração gráfica desta rede em duas dimensões a tornaria de difícil compreensão, com muitos cruzamentos de arcos. Ao invés disso, a elaboração de duas redes forneceu uma visão mais objetiva e menos complicada. Como devemos nos lembrar, o aspecto visual de uma REDE DE PETRI é bastante forte, de modo que o leitor pode tirar algumas conclusões por simples inspeção. Este também é um dos objetivos de uma modelagem utilizando REDES DE PETRI.

Sendo assim podemos visualizar nas figuras 3.4.1 e 3.4.2 as redes construídas para modelar a parte do sistema que cuida da "Atualização dos Preços em Tempo Real" e do "Controle de Estoque em Tempo Real", respectivamente.

Conforme podemos ver o processo GERENTE não faz parte desta modelagem. Este processo é apenas aplicação em banco de dados, não enviando mensagens para nenhum outro processo. Apenas concorre para acesso disciplinado dos arquivos que descrevemos anteriormente. Por este motivo não foi incluído na modelagem.

Então o próximo passo será a validação destas REDES DE PETRI. Para isso utilizamos o programa "ARP - Analisador de REDES DE PETRI", desenvolvido especialmente para este propósito.

Para que se possa analisar uma REDE DE PETRI devemos editar um arquivo texto comum contendo a descrição da rede. Este arquivo pode ser editado através do próprio editor acessado através da execução do programa ou através de algum editor de texto (o edit do DOS, por exemplo). O formato no qual este arquivo deve ser escrito é abordado nas referências que tratam do assunto e que são fornecidas na bibliografia.

Deste modo apresentamos as listagens dos dois arquivos descrevendo as duas redes mostradas nas listagens 1 e 2. Devido ao fato de que as denominações dadas aos lugares ou transições devem respeitar o limite máximo de vinte bytes, alguns nomes aparecem abreviados; apesar disso procuramos seguir as mesmas denominações mostradas nas figuras.

Em posse destes arquivos submetemos à análise do ARP. Lembre-se de que as propriedades gerais devem ser atendidas para a validação de uma rede genérica. Nas listagens 3 e 4 apresentadas nas páginas seguintes temos o resultado da análise do ARP, extraída diretamente do arquivo gerado pelo programa. A análise é auto-explicativa e, como podemos notar, nos mostra que as duas redes analisadas atendem às propriedades gerais e que não foram detectados "live-locks" ou "dead-locks" nas mesmas. Sendo assim as redes foram validadas e demonstram que a modelagem, se seguida corretamente, não deverá causar bloqueio dos processos (propriedade viva limitada) ou problemas de estouro do "buffer" de mensagens (propriedade limitada atendida).

Com isso terminamos a fase de modelagem. Temos em mãos um modelo validado que mostra o caminho que deve ser seguido para a implementação da solução com sucesso. Sem dúvida este é um passo importante para a solução de qualquer problema real e, em específico, para o nosso problema.

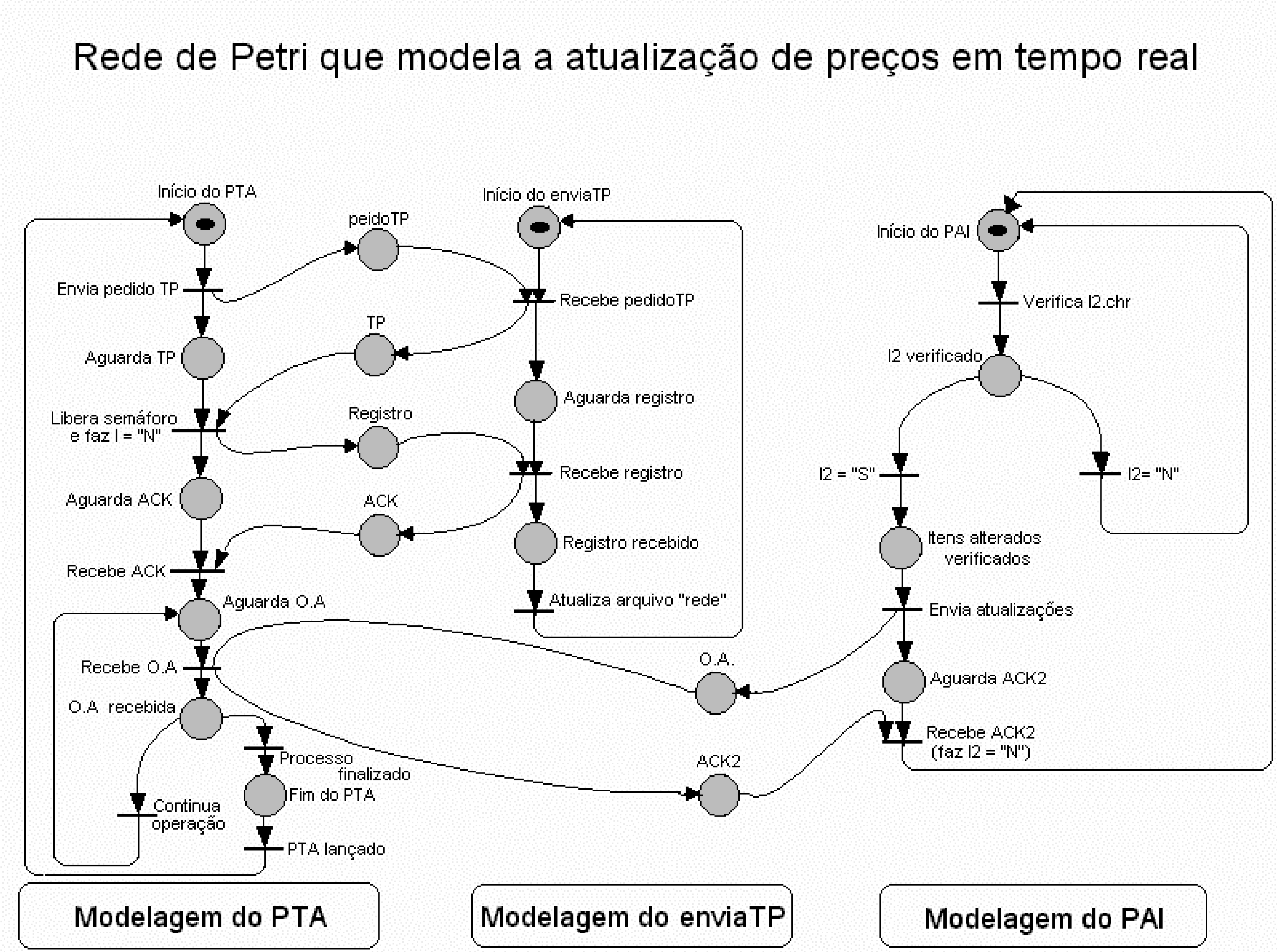


Figura 3.4.1 - REDE DE PETRI que modela a atualização de preços em tempo real

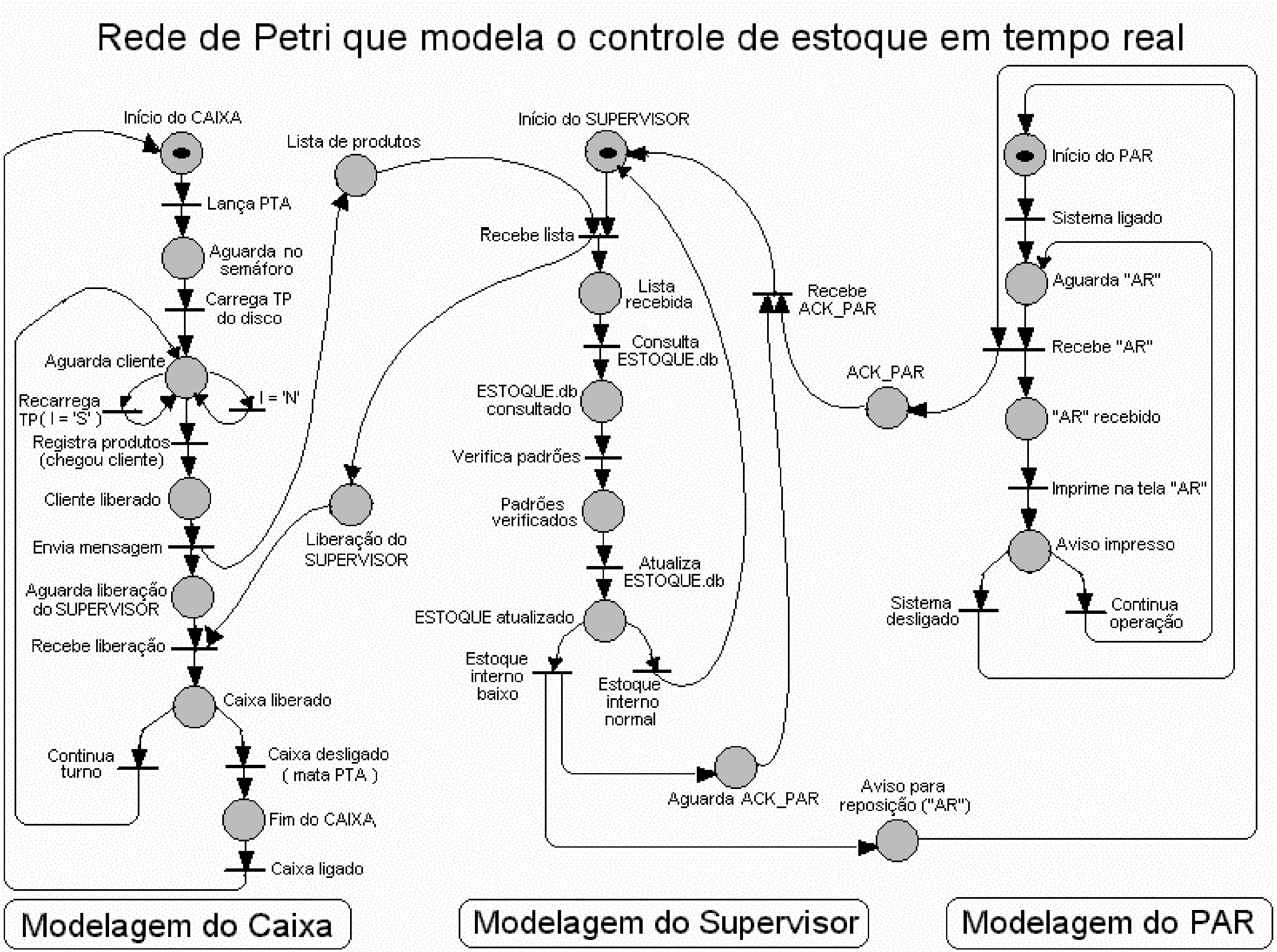


Figura 3.4.2 - REDE DE PETRI que modela a atualização de estoque em tempo real

```
Rede Atualizadora_Precos;{ * Modela a atualizacao de precos em tempo real * }

NODOS
      { * Marca inicial da rede * }
      inicio_PTA , inicio_enviaTP , inicio_PAI                : lugar(1) ;

      { * Lugares comuns das tres redes * }
      pedido_TP , TP , registro , ACK                         :   lugar ;
      ordem_atualizacao , ACK2                                :   lugar ;

      { * Lugares da rede PTA * }
      aguarda_TP , aguarda_ACK , aguarda_OA                   :   lugar ;
      OA_recebida , fim_PTA                                   :   lugar ;

      { * Lugares da rede enviaTP * }
      aguarda_registro , registro_recebido                    :   lugar ;

      { * Lugares da rede PAI * }
      I2_verificado , itens_verificados , aguarda_ACK2        :   lugar ;

      { * Transicoes da rede PTA * }
      envia_pedido_TP , libera_semaforo , recebe_ACK          : transicao;
      recebe_OA  , continua_operacao , processo_finalizado    : transicao;
      PTA_lancado                                             : transicao;

      { * Transicoes da rede enviaTP * }
      recebe_pedido_TP , recebe_registro , atualiza_conf_rede : transicao;

      { * Transicoes da rede PAI * }
      verifica_I2 , I2_S , I2_N , envia_atualizacoes          : transicao;
      recebe_ACK2                                             : transicao;

ESTRUTURA

      envia_pedido_TP :
      ( inicio_PTA            ) ,
      ( pedido_TP , aguarda_TP ) ;

      libera_semaforo :
      ( TP , aguarda_TP        ) ,
      ( registro , aguarda_ACK ) ;

      recebe_ACK :
      ( aguarda_ACK , ACK ) ,
      ( aguarda_OA        ) ;

      recebe_OA :
      ( aguarda_OA , ordem_atualizacao ) ,
      ( OA_recebida , ACK2             ) ;

      continua_operacao :
      ( OA_recebida ) ,
      ( aguarda_OA  ) ;
```

```
processo_finalizado :
( OA_recebida ) ,
( fim_PTA     ) ;

PTA_lancado :
( fim_PTA    ) ,
( inicio_PTA ) ;

recebe_pedido_TP :
( inicio_enviaTP , pedido_TP       ) ,
( aguarda_registro , TP            ) ;

recebe_registro :
( aguarda_registro , registro ) ,
( registro_recebido , ACK     ) ;

atualiza_conf_rede :
( registro_recebido ) ,
( inicio_enviaTP    ) ;

verifica_I2 :
( inicio_PAI        ) ,
( I2_verificado     ) ;

I2_S :
( I2_verificado     ) ,
( itens_verificados ) ;

I2_N :
( I2_verificado     ) ,
( inicio_PAI        ) ;

envia_atualizacoes :
( itens_verificados               ) ,
( ordem_atualizacao , aguarda_ACK2 ) ;

recebe_ACK2 :
( aguarda_ACK2 , ACK2 ) ,
( inicio_PAI          ) ;


FIM.
```

Listagem 1 - Arquivo no formato "ARP" descrevendo a rede Atualizadora_Preços

```
Rede Controladora_Estoque; { * Modela o controle de estoque em tempo real * }

NODOS
      { * Marca inicial * }
      inicio_CAIXA  ,  inicio_SUPERVISOR  ,  inicio_PAR          : lugar(1);

      { * Lugares comuns das tres redes * }
      lista_produtos , liberacao , aviso_reposicao , ACK_PAR         : lugar;

      { * Lugares da rede CAIXA * }
      aguarda_semaforo , aguarda_cliente , cliente_liberado            : lugar;
      aguarda_liberacao , caixa_liberado , fim_CAIXA                   : lugar;

      { * Lugares da rede SUPERVISOR * }
      lista_recebida , ESTOQUE_consultado , padroes_verificados : lugar;
      aguarda_ACK_PAR , ESTOQUE_atualizado                        : lugar;

      { * Lugares da rede PAR * }
      aguarda_AR , AR_recebido , aviso_impresso                       : lugar;


      { * Transicoes da rede CAIXA * }
      lanca_PTA  ,  carrega_TP  ,  i_N  ,  recarrega_TP              : transicao;
      registra_produtos , envia_mensagem , recebe_liberacao       : transicao;
      caixa_desligado , caixa_ligado , continua_turno               : transicao;

      { * Transicoes da rede SUPERVISOR * }
      recebe_lista , consulta_ESTOQUE , verifica_padroes            : transicao;
      atualiza_ESTOQUE, estoque_int_baixo, estoque_int_normal : transicao;
      recebe_ACK_PAR                                                  : transicao;

      { * Transicoes da rede PAR * }
      sistema_ligado , recebe_AR , imprime_aviso                      : transicao;
      continua_operacao  , sistema_desligado                          : transicao;


ESTRUTURA

      lanca_PTA  :
      ( inicio_CAIXA     ) ,
      ( aguarda_semaforo ) ;

      carrega_TP :
      ( aguarda_semaforo ) ,
      ( aguarda_cliente  ) ;

      i_N :
      ( aguarda_cliente  ) ,
      ( aguarda_cliente  ) ;

      recarrega_TP :
      ( aguarda_cliente  ) ,
      ( aguarda_cliente  ) ;
```

```
registra_produtos :
( aguarda_cliente  ) ,
( cliente_liberado ) ;

envia_mensagem :
( cliente_liberado                  ) ,
( lista_produtos , aguarda_liberacao ) ;

recebe_liberacao :
( aguarda_liberacao , liberacao ) ,
( caixa_liberado                ) ;

continua_turno :
( caixa_liberado    ) ,
( aguarda_cliente   ) ;

caixa_desligado :
( caixa_liberado    ) ,
( fim_CAIXA         ) ;

caixa_ligado :
( fim_CAIXA         ) ,
( inicio_CAIXA      ) ;

recebe_lista :
( inicio_SUPERVISOR , lista_produtos ) ,
( liberacao , lista_recebida         ) ;

consulta_ESTOQUE :
( lista_recebida     ) ,
( ESTOQUE_consultado ) ;

verifica_padroes :
( ESTOQUE_consultado  ) ,
( padroes_verificados ) ;

atualiza_ESTOQUE :
( padroes_verificados  ) ,
( ESTOQUE_atualizado   ) ;

estoque_int_normal :
( ESTOQUE_atualizado   ) ,
( inicio_SUPERVISOR    ) ;

estoque_int_baixo :
( ESTOQUE_atualizado               ) ,
( aguarda_ACK_PAR , aviso_reposicao ) ;

recebe_ACK_PAR :
( ACK_PAR , aguarda_ACK_PAR ) ,
( inicio_SUPERVISOR         ) ;
```

```
    sistema_ligado :
    ( inicio_PAR    ) ,
    ( aguarda_AR    ) ;

    recebe_AR :
    ( aguarda_AR , aviso_reposicao ) ,
    ( AR_recebido , ACK_PAR         ) ;

    imprime_aviso :
    ( AR_recebido    ) ,
    ( aviso_impresso ) ;

    continua_operacao :
    ( aviso_impresso ) ,
    ( aguarda_AR     ) ;

    sistema_desligado :
    ( aviso_impresso ) ,
    ( inicio_PAR     ) ;

FIM.
```

Listagem 2 - Arquivo no formato "ARP" descrevendo a rede Controladora_Estoque

```
Enumeracao de estados: rede Atualizadora_Precos (70 estados acessiveis).

Propriedades verificadas:
*-----------------------------------------------------------------------*
A rede em analise e' binaria.
  Lugares Nulos (M = 0): { }
  Lugares Binarios     : {todos(as)}
  Lugares k-Limitados  : { }
  Lugares Nao Limitados: { }

A rede em analise nao e' estritamente conservativa.

A rede em analise e' viva.
  Tr. vivas         : {todos(as)}
  Tr. quase-vivas   : {todos(as)}
  Tr. nao disparadas: { }

A rede e' reiniciavel.

Nao foram detectados "live-locks" na rede.

Nao foram detectados "dead-locks" na rede.
*-----------------------------------------------------------------------*
```

Listagem 3 - Arquivo contendo a análise do "ARP" para a rede Atualizadora_precos

```
Enumeracao de estados: rede Controladora_Estoque (224 estados acessiveis).

Propriedades verificadas:
*-----------------------------------------------------------------------*
A rede em analise e' binaria.
  Lugares Nulos (M = 0): { }
  Lugares Binarios     : {todos(as)}
  Lugares k-Limitados  : { }
  Lugares Nao Limitados: { }

A rede em analise nao e' estritamente conservativa.

A rede em analise e' viva.
  Tr. vivas         : {todos(as)}
  Tr. quase-vivas   : {todos(as)}
  Tr. nao disparadas: { }

A rede e' reiniciavel.

Nao foram detectados "live-locks" na rede.

Nao foram detectados "dead-locks" na rede.
*-----------------------------------------------------------------------*
```

Listagem 4 - Arquivo contendo a análise do "ARP" para a rede Controladora_estoque

3.5) - Comentários

Neste capítulo descrevemos o problema a ser resolvido e as exigências feitas para a implementação. Como foi visto, apesar de hipotética, a situação poderia ser perfeitamente real. Logo depois modelamos a solução utilizando os conceitos teóricos estudados no capítulo 2, através da divisão das tarefas entre vários processos em um sistema multitarefa. Detalhamos qual a tarefa que cada processo em particular desempenharia, e modelamos estes processos por REDES DE PETRI (assunto cuja base teórica foi superficialmente abordada). Estas redes foram analisadas e validadas pelo “software” ARP (desenvolvido especialmente para análise de REDES DE PETRI) com sucesso.

Em particular, a validação das REDES DE PETRI através do ARP pode garantir que os programas que serão desenvolvidos seguindo esta modelagem não apresentarão bloqueios individuais ou coletivos e nem sobrecarga da rede física. Sem dúvida a validação de uma modelagem é bastante útil e importante.

O próximo passo, então, será a implementação, em linguagem ANSI C, de todos os processos modelados. Esta implementação deverá seguir o modelo construído neste capítulo e incluir todos os mecanismos necessários para permitir acesso concorrido aos arquivos, comunicação entre os processos, lançamento de processos, entre outros. O próximo capítulo trata exatamente do código que implementa a solução proposta e modelada neste. De fato, isto é o que resta para para findar nosso projeto.

Capítulo 4 - Estudo de Caso : Implementação e Resultados

4.1) - Introdução

Seguindo a modelagem descrita no capítulo anterior vamos comentar, agora, a nossa implementação em código ANSI C. Para cada processo criamos um programa fonte com nome idêntico ao associado na modelagem. A idéia foi a de, **seguindo diretamente as REDES DE PETRI**, elaborar os códigos relativos a cada processo que estará executando no sistema operacional LINUX.

Este capítulo está dividido, basicamente, em oito partes. Cada processo será comentado em separado. Não abordaremos em detalhes cada programa. Todos juntos somam mais de três mil linhas, e isto seria exaustivo e sem sentido. Procuraremos, apenas, fornecer uma visão global e situar o leitor acerca das técnicas que foram utilizadas para a implementação e qual a relação com os exemplos descritos no capítulo 2 desta documentação. Quando necessário iremos apresentar pequenos trechos dos códigos para ilustrar e auxiliar na explanação feita. Também vamos esclarecer qual a função dos arquivos que são criados durante a execução destes programas e como eles se interagem com os demais processos. Apesar de os programas conterem documentação própria razoavelmente satisfatória, a leitura deste capítulo pode auxiliar o leitor na tarefa de entender a solução implementada.

4.2) - O processo CAIXA

O programa desenvolvido para implementar este processo se apresenta sob o título de “caixa.c”. Este é um dos processos mais extensos em termo de linhas de código. De fato este processo é o que lida com a interface de apresentação ao usuário, fato que pode explicar, em parte, o tamanho do código. Vale a pena ressaltar que utilizamos a biblioteca “curses.h” para a apresentação da interface.

A nossa implementação procurou seguir a modelagem da maneira mais fiel possível. A primeira tarefa realizada é o lançamento do processo PTA em “BACKGROUND”. A técnica utilizada segue o exemplo de “lançamento de processos a partir de um processo em execução” estudado no capítulo 2. Vejamos qual é o pedaço do código que realiza este lançamento :

```
// lanca PTA relativo a este CAIXA
pid = fork( );
if (pid == 0)
{
   execl(PTA,PTA,"&", NULL);
   _exit(EXIT_FAILURE);
}
else if (pid>0) printf("PTA lançado com sucesso. (PID = %d)\n", pid);
else if (pid < 0)
{
   printf("Erro no lançamento do processo PTA\n");
   printf("Execução suspensa. Contate o fabricante. \n");
   exit(1);
}
```

Logo após o CAIXA aguarda no semáforo, que neste caso funciona como um indicador de que se deve carregar a tabela de preços que está no arquivo de disco "listprod" na memória. A implementação do semáforo segue, exatamente, o exemplo estudado que aborda este assunto. É feito um acesso disciplinado a este arquivo através de um semáforo e carrega-se a tabela de preços em uma lista duplamente encadeada na memória (isto é realizado pela função "load( )"). Após isto o semáforo é liberado para que o PTA possa atualizar a tabela de preços, quando necessário. Isto é feito pelas seguintes linhas de código :

```
// aguarda no semáforo para acesso disciplinado e carrega TP do disco
sleep(1);
if ( (semid = initsem(semkey)) < 0 )
        exit(1);
printf("processo CAIXA aguardando sinal para carregar TP…\n");
down(semid);
sleep(2);
printf("Processo CAIXA carregando TP na memória…\n");
sleep(2);

// carrega TP na memoria em uma lista encadeada
inicio=ultimo=NULL;
load( );
```

```
printf("Processo CAIXA liberando arquivo para atualizacoes…\n");
sleep(2);
//libera semáforo
up(semid);
```

É, então, organizada a tela de interface gráfica, tarefa realizada pela função **"tela_inicial( )"**. A partir daí fazemos uso de funções que simulam o comportamento de um sistema real. Entretanto não nos preocupamos em descrever o fluxo de clientes ou de produtos comprados através de algum modelo matemático, embora isto devesse ser feito se esta simulação tivesse que produzir resultados a serem analisados sob o prisma de desempenho do sistema. Entretanto estamos preocupados, única e exclusivamente, com o aspecto relacionado ao sistema operacional. Por isso utilizamos a função **"sorteia_numero(int limite_superior)",** que retorna um número inteiro aleatório situado entre um e o limite_superior, diversas vezes na nossa simulação.

A primeira função que simula o comportamento de um CAIXA real é a **"aguarda_cliente( )".** Nesta função o intervalo de tempo até a próxima chegada de um cliente é definida pela função "sorteia_número". Em cada segundo que se espera o processo verifica o arquivo "I.chr". Caso este arquivo contenha o valor "S" (indicando que foram feitas atualizações na tabela de preços), então há o recarregamento do arquivo "listprod" na mesma lista encadeada que continha a lista desatualizada, substituindo-a.

Após a chegada de um cliente genérico utilizamos a função "atende_cliente( )" que determina, aleatoriamente, quantos e quais produtos serão comprados. Também é aleatório o tempo necessário para registrar este produto, variando de um a três segundos. Enquanto os produtos vão sendo registrados, também vão sendo armazenados em uma segunda lista encadeada na memória. Esta lista encadeada guardará, apenas, as informações dos produtos comprados pelo cliente corrente; quando do atendimento do próximo cliente, esta lista será substituída.

Após o atendimento do cliente grava-se esta lista corrente em um arquivo em disco denominado "COMPRAS", cujo acesso não necessita ser disciplinado, já que este processo é o único que lida com ele. Este arquivo conterá, sempre, a lista de compras do cliente corrente, assim como a lista encadeada. Este arquivo poderia ser utilizado caso ocorresse algum erro na execução ou corte de luz e fosse necessário enviar os dados do último cliente atendido. Apesar disso este arquivo não é mais utilizado pelo nosso processo.

Então acionamos a função **"prepara_mensagem( )"**, que têm acesso à lista encadeada com a lista de produtos comprados pelo cliente corrente e gera uma mensagem que é do seguinte formato :

[tamanho da mensagem][P][código do produto,quantidade comprada]~[código do produto,quantidade comprada]~…|

Tendo a mensagem em mãos, a função **"envia_mensagem( )"** trata de abrir conexão com o SUPERVISOR na porta "49000" e enviar a mensagem. A implementação do SOCKET para comunicação com o processo SUPERVISOR é realizada de forma idêntica à utilizada no exemplo de comunicação de processos estudado no capítulo 2. É útil ressaltar que o processo tenta abrir conexão até obter sucesso nesta operação, esperando cinco segundos para cada nova tentativa. Isto significa que, se o terminal SUPERVISOR não estiver em operação, fica impossível o atendimento de um novo cliente, e o CAIXA fica preso em um "loop" sem fim. Abaixo temos um trecho da função **"envia_mensagem()"** que tenta abrir conexão até conseguir obter sucesso nesta operação :

```
do
{
  conexão = open_client-socket(49000);
  if (!conexão)
  {
        atualiza_status("Tentativa de conexao fracassada...           ",2);
        sleep(3);
  }
  else
  {
     atualiza_status("Conexao aberta...                          ");
     sleep(2);
   }
} while(!conexao);
```

A função **"atualiza_status"** é encarregada de mostrar na tela, na posição correta, a mensagem que é passada como primeiro argumento na cor correspondente ao segundo argumento.

Após o estabelecimento da conexão o CAIXA envia a mensagem contendo a lista de produtos comprados e aguarda a confirmação no formato "LB" ou pedido de retransmissão do SUPERVISOR. Se for enviado pedido de retransmissão a mensagem é novamente transmitida. Veja o trecho do código que implementa o protocolo para troca de mensagens :

```
for( ; ;)
{
   atualiza_status("Enviando relatorio para o SUPERVISOR...        ",3);
   write(conexao,mens,strlen(mens)+1);
   sleep(2);
```

```
    atualiza_status("Aguardando sinal de liberação…                    ",4);
    sleep(2);
    read(conexao,temp,3);
    tamanho = atoi(temp);
    read(conexao,temp,tamanho);
    if (!strcmp(temp, "LB"))
    {
        close(conexao);
        return;
    }
    else
    {
        atualiza_status("Pedido de retransmissao enviado…          ",2);
        sleep(2);
    }
}
```

Tendo feito isso o processo pode seguir dois caminhos, dependente de aleatoriedade. Com maior probabilidade há a continuação da operação do CAIXA, que entra em modo de espera de um novo cliente. Entretanto pode haver o fim da execução. Para que isso ocorra é necessário que haja a finalização do PTA outrora lançado. Isto é feito seguindo o exemplo de finalização de processos analisado no capítulo 2. Vejamos a chamada que finaliza o processo PTA :

```
printf("Finalizando processo PTA…\n");
kill(pid,SIGKILL);
```

Com isto termina a implementação do nosso primeiro processo.

4.3) - O processo PTA

O programa desenvolvido para implementar este processo se apresenta sob o título de “pta.c”. A complexidade deste processo está no fato de que lida com dois tipos de conexões, para se comunicar com dois processos diferentes : o enviaTP e o PAI.

Inicialmente ocorre o acionamento do semáforo (o mesmo que é acessado pelo processo CAIXA). A implementação é a mesma utilizada no processo CAIXA, já que este semáforo é o mesmo utilizado pelos dois processos.

Após isso há a tentativa de abertura de conexão com o processo enviaTP na porta “48000”. Estas tentativas prosseguem até que seja a conexão seja efetivada com sucesso, sendo que cada nova tentativa é feita cinco segundos após a anterior. O trecho do código que implementa esta tarefa é a seguinte :

```
// abre conexao com o processo ‘enviaTP’ no terminal SUPERVISOR
do
{
        conexao = open_client_socket(48000);
        if (!conexao) sleep(5); // espera cinco segundos antes de uma nova tentativa
} while(!conexao)
```

Assim que a conexão é estabelecida é enviado pedido de transmissão da tabela de preços, no formato “TP”. Isto é feito através da chamada :

```
write(conexao, “  3TP\0”, 6);
```

É utilizada, então, a função **“recebe_TP”**, que utiliza protocolo de comunicação muito simples, baseado na confirmação dos itens enviados. Esta lista é gravada no arquivo de disco “listprod”.

O semáforo é liberado em seguida (sinalizando que o CAIXA pode acessar o arquivo “listprod”) e é enviado o registro do CAIXA, o qual contêm o número de identificação e a porta para comunicação com o CAIXA corrente (estes dados estão contidos no arquivo “porta”, que é criado pelo programa “config.c”). Na verdade, se estes programas fossem executados realmente em diversas máquinas diferentes, bastaria definir uma única porta lógica para todos os CAIXAS e alterar apenas a identificação IP de acordo com a máquina com a qual fosse se estabelecer conexão. Entretanto, estaremos executando todos os programas em uma só máquina, de modo que a identificação desta porta deve ser diferente de todas as outras utilizadas no sistema, de modo a não haver conflito.

Aguarda-se, então, o recebimento do “acknoledge” do processo enviaTP no formato “RGOK”. No caso do envio de qualquer outra resposta ocorre a retransmissão destas informações. Esta tarefa é realizada pela função **“envia_registro”.** Vejamos como isto é feito no código :

```
do
{
   write(sockfd, mensagem, 13);
   read(sockfd, inicial, 3);
   inicial[3] = '\0';
   resposta[0] = '\0';
   read(sockfd, resposta, (atoi(inicial)) );
 } while(strcmp(resposta, "RGOK\0") );
```

Após isso a conexão com o processo enviaTP é fechada. Como pode se notar, nesta primeira parte o processo PTA faz papel de CLIENTE, enquanto o processo enviaTP, que executa remotamente, faz papel de SERVIDOR. O PTA só estabelece conexão com o enviaTP quando do início de sua execução. Nesta implementação utilizamos, exatamente, o método descrito no exemplo estudado no capítulo 2 para troca de mensagens utilizando SOCKETS.

Então, neste ponto, há uma inversão de papéis. O PTA entra no modo de escuta na porta lida do arquivo “porta” do disco, aguardando pedidos de requisição. Claramente, agora, o PTA faz papel de SERVIDOR. Só que desta vez não foi utilizada a técnica de duplicação do processo para o atendimento de novas conexões, já que as conexões serão sempre requisitadas pelo mesmo processo, o PAI (não se caracteriza acesso concorrido). Assim, após o estabelecimento da conexão, o PTA recebe as ordens de atualização até que se verifique a ocorrência da mensagem **“F”**, indicando fim de envio. Para cada ordem de atualização recebida acionamos a função **“atualiza_TP”,** que acessa o arquivo “listprod” de maneira disciplinada e atualiza os dados recebidos na posição correta. O envio do “acknoledge2” no formato “DC” (desconectando), encerra a comunicação com o PAI e a conexão é fechada.

Após isso o estado do arquivo “I.chr” é alterado para “S”, sinalizando para o CAIXA que há a necessidade de recarregar a tabela de preços do disco. Esta operação também exige acesso disciplinado. Então o processo entra em modo de escuta para o recebimento de futuras ordens de atualização. O “loop”que implementa o atendimento da conexão é o seguinte :

```
for( ; ; )
{
  // abre conexao (socket) para receber ordens de atualizacao
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *), &cli_addr, &cliclen);
  if (newsockfd<0)
        perror("server : accept error");
  do
  {
        //aguarda ordem de atualizacao
        read(newsockfd, inicial, 3);
        inicial[3] = '\0';
        tam_mens = atoi(inicial);
        buffer[0] = '\0';
        read(newsockfd, buffer, tam_mens); // le ordem de atualizacao
        if (buffer[0] == 'F')
                write(newsockfd, "  3DC\0", 6);
        else
        {
                atualiza_TP(buffer);
                write(newsockfd, "  3OK\0", 6);
        }
   } while(buffer[0] != 'F');
   shutdown(newsockfd,2);
   down(semid);
   faz_I("S\0");
   up(semid);
}
```

Para que possa haver o funcionamento correto do CAIXA e do PTA é necessário executar o aplicativo "./config" que cria o arquivo "porta". Conforme já comentado, devido ao fato de que estaremos executando todos os programas em uma única máquina, cada CAIXA terá uma identificação e uma porta particulares e diferentes de todos os outros CAIXAS. Supõe-se que o instalador execute este aplicativo antes de conectar novos terminais na rede. Só é preciso executá-lo na primeira vez que o CAIXA entra em operação, a não ser que se deseje mudar a porta de comunicação.

Sendo assim termina a implementação do segundo processo, que executará em "BACKGROUND" nos terminais CAIXA.

4.4) - O processo GERENTE

O programa desenvolvido para implementar este processo se apresenta sob o título de "gerente.c". Este é o programa mais extenso concentrando, aproximadamente, um terço de toda a solução. A maior parte do código repousa sobre o tratamento da interface de apresentação para o usuário. Estas rotinas são bastante extensas e o nosso objetivo não é o de comentá-las aqui. Algumas outras rotinas tratam do acesso ao banco de dados através da utilização da biblioteca "ndbm.h", uma facilidade do ANSI C. Também não vamos comentar como este acesso é realizado. Basta saber que o acesso aos arquivos é feito de forma disciplinada através de um conjunto de dois semáforos que é criado. Esta implementação de semáforos difere do exemplo desenvolvido no capítulo 2, que cria apenas um conjunto contendo um semáforo. Por este motivo desenvolvemos uma outra biblioteca chamada **"semaphores2",** que implementa as primitivas que manipulam um conjunto de dois semáforos. A sua forma é bastante semelhante à da biblioteca utilizada anteriormente (**"semaphores.h"**). Basicamente são alterados os parâmetros que determinam a quantidade de semáforos que estarão contidos no conjunto criado. A chamada abaixo apresenta diferenças em relação às utilizadas na biblioteca anterior :

if ((semid = semget(semkey, 2, SEMPERM|IPC_CREAT|IPC_EXECL)) == -1)…

Além disso as implementações do UP( ) e DOWN( ) admitem a passagem do argumento indicando a que semáforo estamos nos referindo. O conteúdo dos arquivos que serão acessados através destes dois semáforos foi discutido no capítulo anterior.

O processo GERENTE inicialmente lança os outros três processos que estarão executando em "BACKGROUND" no terminal SUPERVISOR. Estes lançamentos são feitos de forma análoga às maneiras já estudadas, descritas e utilizadas.

A partir daí as tarefas a serem realizadas são definidas pelo próprio usuário, que escolhe opções em um "menu" e entra com os dados relativos a cada função que escolhe. Não há comunicação deste processo utilizando SOCKETS com nenhum outro. A situação mais interessante é atingida ao se liberar algum lote de atualizações para que o PAI possa enviá-lo para todos os CAIXAS. Isto é feito alterando-se o estado do arquivo "I2.chr" para "S", o que é realizado pela função **"I2S"** que utiliza acesso disciplinado. Em cada registro da tabela de preços temos um campo especial que determina se o registro foi alterado ou não (indicado pelas letras “S” e “N”, respectivamente).

A execução termina com a escolha da última opção do “menu”. Após isso, todos os processos em “BACKGROUND” são finalizados e o funcionamento de todo o sistema fica impossibilitado (situação que é advertida no pedido de confirmação da escolha). O método de finalização é o mesmo já exemplificado (através da primitiva “KILL( )”).

Sendo assim termina a implementação do terceiro processo, que tem destacada importância no sistema. A interface para interação com o usuário é bastante agradável, como pode ser visto através de sua execução digitando “./gerente”.

4.5) - O processo enviaTP

O programa desenvolvido para implementar este processo se apresenta sob o título de “enviaTP.c”. Este é um dos mais simples em termo de mecanismo e de tamanho de código. Basicamente este processo é um SERVIDOR que permite conexão na porta “48000”. Aqui utilizamos a técnica de duplicação através do uso do “FORK( )” para atender a um novo pedido de requisição (exatamente como no exemplo do capítulo 2). Isto porque pode haver uma situação em que um PTA esteja conectado e algum outro PTA pertencente a outro CAIXA requisite conexão (situação de acesso concorrido). Entretanto, como o acesso ao arquivo “TP.db” é disciplinado, enquanto uma cópia não terminar a transferência da tabela outras cópias devem aguardar no semáforo para acessar este arquivo disciplinadamente e realizar as suas transferências.

Inicialmente o enviaTP obtém a identificação do conjunto de semáforos criados e inicializa o arquivo de configuração da rede com um primeiro registro nulo (esta tarefa é realizada pela função **“inicializa_conf”**). Toda vez que este processo é executado do início ele trunca este arquivo, apagando os dados de eventuais CAIXAS que estavam conectados quando de sua finalização anterior.

Em seguida o processo entra em modo de escuta na porta citada e atende às conexões requeridas por eventuais processos PTA. Isto é feito através das seguintes linhas :

```
// atende aos processos PTA que enviam pedido de transmissão da TP
sockfd = open_server_socket(48000);
trata_conexao(sockfd);
exit(0);
```

Ao estabelecer uma conexão há a transferência da tabela de preços para o PTA conectado (realizada pela função **“envia_TP”**). Como já sabemos, após o fim da transferência, este processo recebe uma mensagem contendo a identificação do CAIXA e a porta de comunicação proprietária. Em posse destes dados o processo atualiza o arquivo de configuração da rede (“rede”). Ele deve verificar se as informações relativas ao CAIXA em questão já foram registradas anteriormente, de modo que seria necessário apenas a atualização. Esta situação pode ocorrer quando se desliga algum CAIXA por um intervalo de tempo e depois ocorre um religamento. A função **“registra_CAIXA”** realiza estas tarefas utilizando, também, a função **“valida_caixa”**.

Após enviar a mensagem de liberação no formato “RGOK”, a conexão é fechada e o processo cópia termina sua execução através da chamada “exit(0)” (presente no trecho anterior de código), assim como no último exemplo discutido no capítulo 2.

O entendimento deste processo é tarefa fácil, principalmente se foi entendido o exemplo de comunicação entre processos descrito no capítulo 2.

Desta forma ficou implementado o nosso quarto processo.

4.6) - O processo PAI

O programa desenvolvido para implementar este processo se apresenta sob o título de “pai.c”. A exemplo do processo enviaTP, esta implementação também é bastante simples em termo de mecanismo e de tamanho de código. Aqui estamos lidando com um CLIENTE, que deve se conectar a vários SERVIDORES para enviar mensagens contendo as atualizações que foram feitas pelo processo GERENTE.

Este processo é formado por um “loop” infinito, onde a cada cinco segundos verifica-se o conteúdo do arquivo “I2.chr”. Como este arquivo também pode acessado pelo GERENTE, então esta consulta deve ser feita através de acesso disciplinado no primeiro semáforo do conjunto criado. Caso este arquivo contenha o estado “S”, isso indicará que existem alterações a serem transmitidas. Antes de enviar as atualizações, o processo substitui o estado do arquivo “I2.chr” para “N”. Este “loop” é implementado pelas seguintes linhas de código :

```
for( ; ; )
{
   down(semid,0);
   if (check_sem( ) == 1) // Gerente sinalizou que existem alteracoes
          atualiza_caixas( ); // envia atualizacoes para todos os PTA’s
   up(semid,0);
   sleep(5); // aguarda 5 segundos antes de consultar novamente I2.chr
}
```

Todos os registros que forem alterados no arquivo “TP.db” conterão estado “S” no campo de "status" do registro.

Dois pontos devem ser comentados aqui. Primeiro devemos consultar o arquivo “rede” e verificar as portas às quais devemos tentar abertura de conexão. Estas portas, conforme já foi visto, correspondem aos processos PTAs que estão em modo de escuta. Agora imagine que, por algum problema ou anormalidade, algum CAIXA seja desligado ou tenha execução interrompida. Neste caso não há como avisar o processo enviaTP de que tal CAIXA não está em operação. Por outro lado, tendo em vista que o único processo que requisita conexão com o PTA é este, se houver o pedido de conexão e o PTA estiver em execução o pedido é imediatamente aceito. Por exclusão, podemos concluir que se o pedido não for atendido, então o PTA relativo à porta está fora de operação. Baseando-se nesta conclusão, o processo PAI tenta abrir conexão em cada porta três vezes. Caso não obtenha sucesso, então há a atualização do arquivo “rede”, invalidando as informações relativas aquele CAIXA. Isto é feito através da função **“atualiza_caixas”.**

Em segundo lugar, após aberta uma conexão, devemos consultar todos os registros do arquivo da tabela de preços e verificar quais foram alterados. Para cada registro alterado enviamos uma mensagem com os novos dados (isto é feito através da função **“envia_atualizações”**). De fato alguém pode argumentar que, para listas de preços extensas e um número de CAIXAS grande conectados esta tarefa pode demandar tempo e prejudicar a eficiência do sistema. Este argumento é válido, desde que estivéssemos preocupados com aspectos de implementação para uso real deste projeto. Entretanto, estamos preocupados apenas com aspectos de didática em termos de sistema operacional, de modo que esta técnica é válida. Um modo de solucionar este problema, se fosse realmente necessário, seria a criação de um arquivo temporário que conteria apenas as últimas atualizações feitas. Então o processo PAI só necessitaria consultar este arquivo para transferir estas atualizações. Apesar disso, a velocidade de acesso das máquinas vendidas atualmente não torna crítico o método que nós utilizamos.

Deste maneira implementamos o nosso quinto processo.

4.7) - O processo SUPERVISOR

O programa desenvolvido para implementar este processo se apresenta sob o título de “supervisor.c”. A exemplo do enviaTP, temos aqui um SERVIDOR que aguarda pedidos de conexão dos processos CAIXA na porta “49000”. A cada pedido de conexão há a duplicação do processo para o atendimento do respectivo CAIXA (de maneira análoga a que já exemplificamos e utilizamos). Este atendimento é realizado pela função **“trata_conexao”**. Após o recebimento da mensagem contendo a lista com os códigos e as quantidades dos produtos que foram comprados e o envio da liberação, há uma atualização do banco de dados através de acesso disciplinado no segundo semáforo do conjunto criado. Além disso o SUPERVISOR verifica se há a necessidade de reposição interna nas gôndolas (se houver então autoriza, enviando mensagem para o PAR e atualizando o banco de dados através da função **"avisa_par"**) e se o produto está em falta no depósito. A lista com todos os produtos que estão em falta no depósito pode ser consultada através de uma das opções do "menu" principal do processo GERENTE. A implementação para a utilização dos SOCKETS e dos semáforos utilizados é a mesma utilizada no exemplo desenvolvido no capítulo 3.

Desta maneira terminamos os comentários acerca dos processos em execução no terminal SUPERVISOR. Com certeza neste terminal está a maior concentração de código dentro da nossa solução.

4.8) - O processo PAR

O programa que implementa este processo se apresenta sob o título de "par.c". Sem sombra de dúvida este é o processo mais simples dentre os sete desenvolvidos para implementar a nossa solução. Simplesmente temos um SERVIDOR que fica permanentemente em modo de escuta na porta "47000". Assim que alguma cópia do processo SUPERVISOR se conecta, há a duplicação do PAR para o atendimento. As mensagens contendo o código do produto e a quantidade a ser resposta são recebidas e mostradas de maneira a alertar o operador. Quando os pedidos cessam, a conexão é quebrada. As poucas linhas que implementam este processo são quase uma repetição das técnicas que foram utilizadas no desenvolvimento dos processos anteriores. Por isso não são necessários maiores comentários.

4.9) - Resultados da implementação

Nesta seção final deste capítulo vamos comentar e mostrar alguns resultados da execução dos programas desenvolvidos.

Inicialmente, para que o sistema possa funcionar, devemos executar o processo GERENTE, o que é feito "./gerente". Este programa deve executar em um terminal separado ou em alguma seção aberta em um mesmo terminal. Após as mensagens iniciais o que se verá é exatamente o conteúdo da tela 1 mostrada na página seguinte.

Como se vê podemos escolher uma dentre as 8 opções existentes. Caso se escolha a opção 1 o sistema requisita a entrada dos dados relativos ao produto que se quer incluir. Devemos digitar os dados. A mudança de campo é automática quando se chega ao fim de cada campo. Se os dados não ocuparem todo o espaço disponível devemos teclar <ENTER> para mudar de campo. Após digitar todos os dados a tela que se verá é a que é mostrada na tela 2.

```
Supermercados 171 Ltda     -     CGC  :  32.305.981/0012     -     IE  : 85.456.899
Aplicativo de gerenciamento do controle de estoque        -        Terminal Supervisor

Quarta                                                             26 / 08 / 98

                          ** Menu de opcoes **

( 1 ) - Incluir produto na lista
( 2 ) - Excluir produto da lista
( 3 ) - Altera atributos basicos de um item(descricao ou preco)
( 4 ) - Altera atributos de controle de um item
( 5 ) - Mostra lista de produtos em falta no deposito
( 6 ) - Consulta informacoes a respeito de um item
( 7 ) - Submete as alteracoes feitas para a atualizacao dos CAIXAS
( 8 ) - Encerra execucao

Escolha uma das opcoes acima : (   )
```

Tela 1 : Menu principal do GERENTE

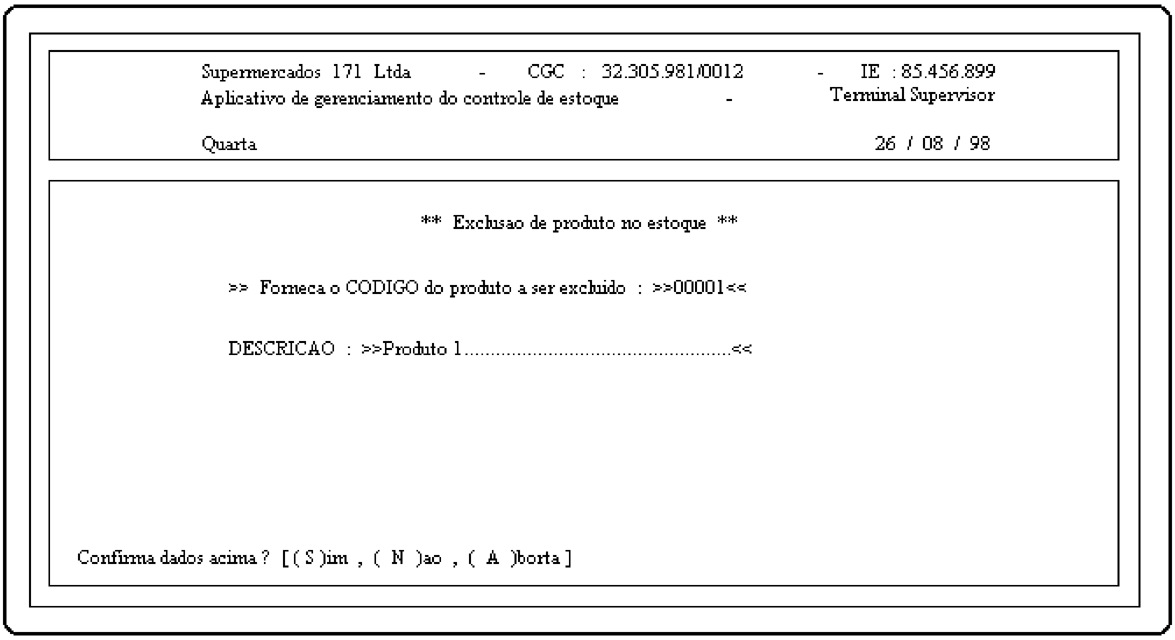
```
Supermercados 171 Ltda     -     CGC  :  32.305.981/0012     -     IE  : 85.456.899
Aplicativo de gerenciamento do controle de estoque        -        Terminal Supervisor

Quarta                                                             26 / 08 / 98

                    ** Inclusao de produto no estoque **

                       >> Atributos basicos do item <<

     CODIGO  : >>00001<<                              PRECO : >>1.00 <<
     DESCRICAO  : >>Produto 1...............................................<<

                       >> Atributos de controle do item <<

          QUANTIDADE  INICIAL NA GONDOLA  : >>1000  <<
          QUANTIDADE MINIMA NA GONDOLA  : >>300   <<
          QUANTIDADE INICIAL NO DEPOSITO  : >>10000<<
          QUANTIDADE MINIMA NO DEPOSITO : >>3000  <<

Confirma dados acima ? [ ( S )im , ( N )ao , ( A )borta ]
```

Tela 2 - Inclusão de um novo produto

Se escolhermos a opção 2 o sistema requisita o código do produto que se quer eliminar do cadastro. Após fornecida esta informação o sistema pede confirmação, situação que é mostrada na tela 3.

Escolhendo a opção 3 vemos que o sistema requisita o código do produto no qual queremos alterar os atributos básicos. Pode-se escolher qual(is) atributo(s) queremos alterar. Quando todas as mudanças tiverem sido realizadas o que se visualizará é a tela 4.

A opção 4 nos leva a um procedimento semelhante a opção 3. A tela 5 nos mostra a situação em que todos os atributos de controle desejáveis foram alterados e o sistema aguarda confirmação.

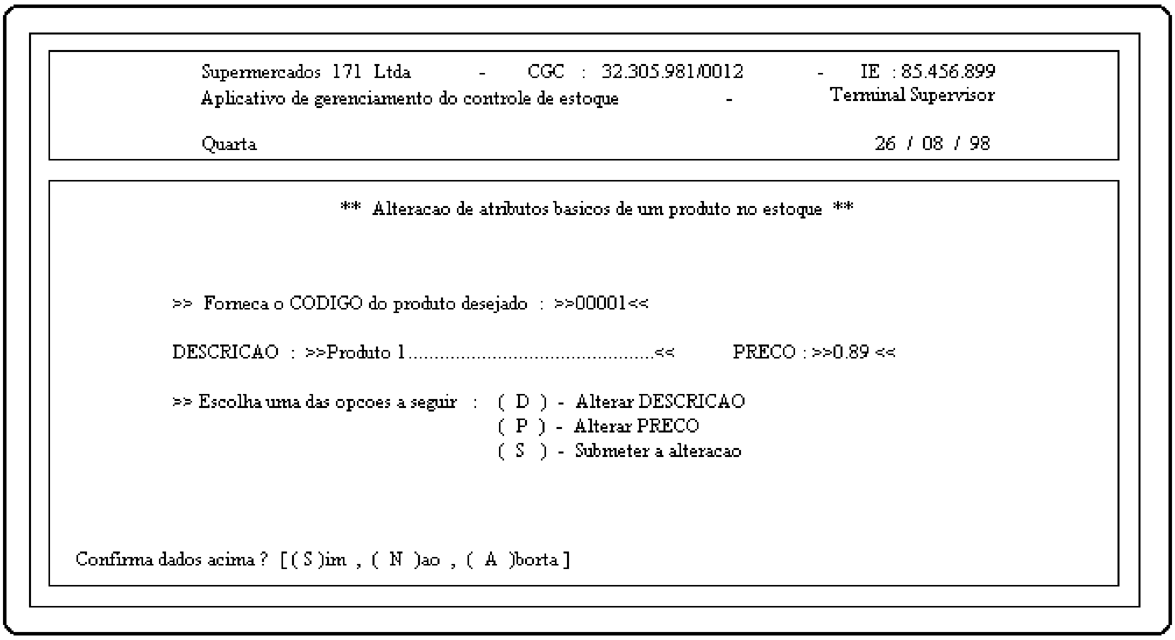


Tela 3 - Exclusão de produto

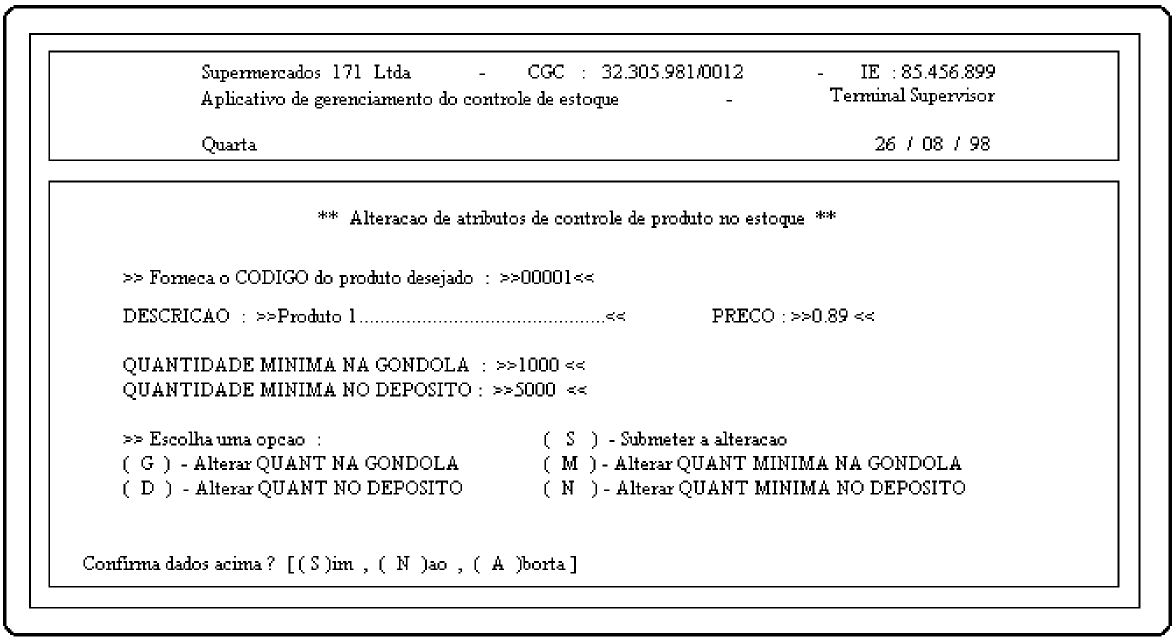
A opção 5 listará os produtos que estão em falta no depósito. Na tela 6 temos a situação em que nenhum produto está em falta no depósito.

Ao escolher a opção 6 podemos consultar todos os dados referentes a um determinado produto. Para isso devemos fornecer o código deste produto. A tela 7 mostra a consulta ao mesmo produto que vem servindo como exemplo.

A opção 7 é a ordem para que as alterações realizadas em um ou mais produtos sejam transmitidas para os caixas que estão conectados à rede. A única mensagem que o sistema emite é mostrada na tela 8.



Tela 4 - Mudança de atributos básicos de um produto



Tela 5 - Mudança de atributos de controle de um produto

Supermercados 171 Ltda - CGC : 32.305.981/0012 - IE : 85.456.899
Aplicativo de gerenciamento do controle de estoque - Terminal Supervisor

Quarta 26 / 08 / 98

** Lista de produtos em falta no deposito **

Lista fornecida. Tecle algo...

Tela 6 - Consulta à lista de produtos em falta no depósito

Supermercados 171 Ltda - CGC : 32.305.981/0012 - IE : 85.456.899
Aplicativo de gerenciamento do controle de estoque - Terminal Supervisor

Quarta 26 / 08 / 98

** Consulta individual de produto cadastrado no estoque **

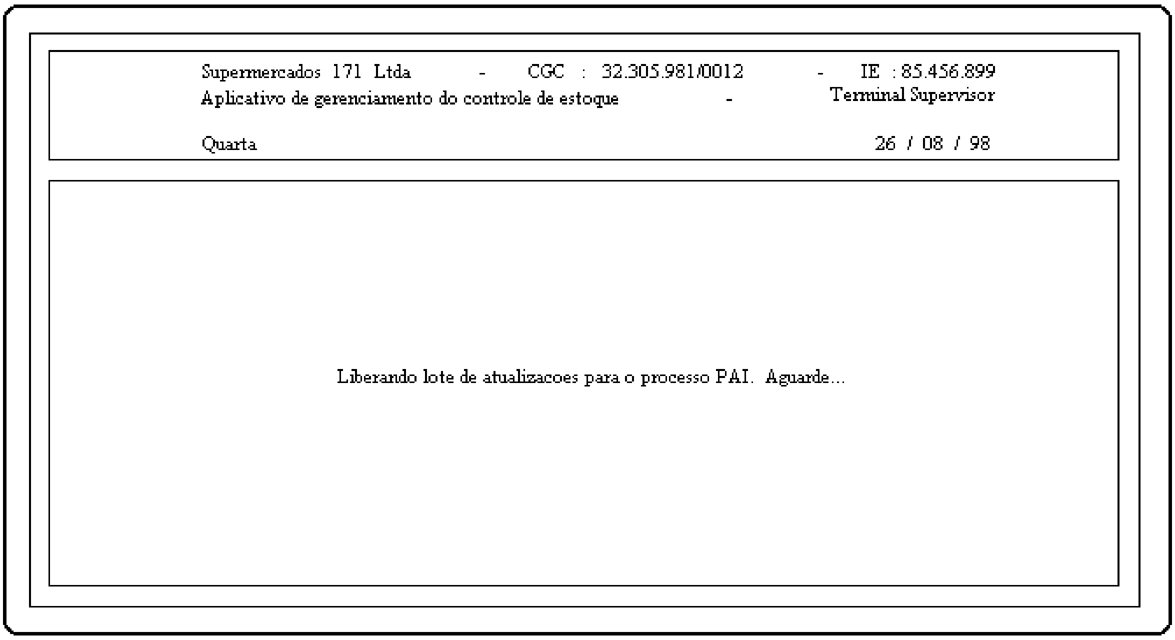
>> Forneca o CODIGO do produto desejado : >>00001<<

DESCRICAO : >>Produto 1..<< PRECO : >>1.00 <<

QUANTIDADE INICIAL NA GONDOLA : >>1000 <<
QUANTIDADE MINIMA NA GONDOLA : >>300 <<
QUANTIDADE INICIAL NO DEPOSITO : >>10000<<
QUANTIDADE MINIMA NO DEPOSITO : >>3000 <<

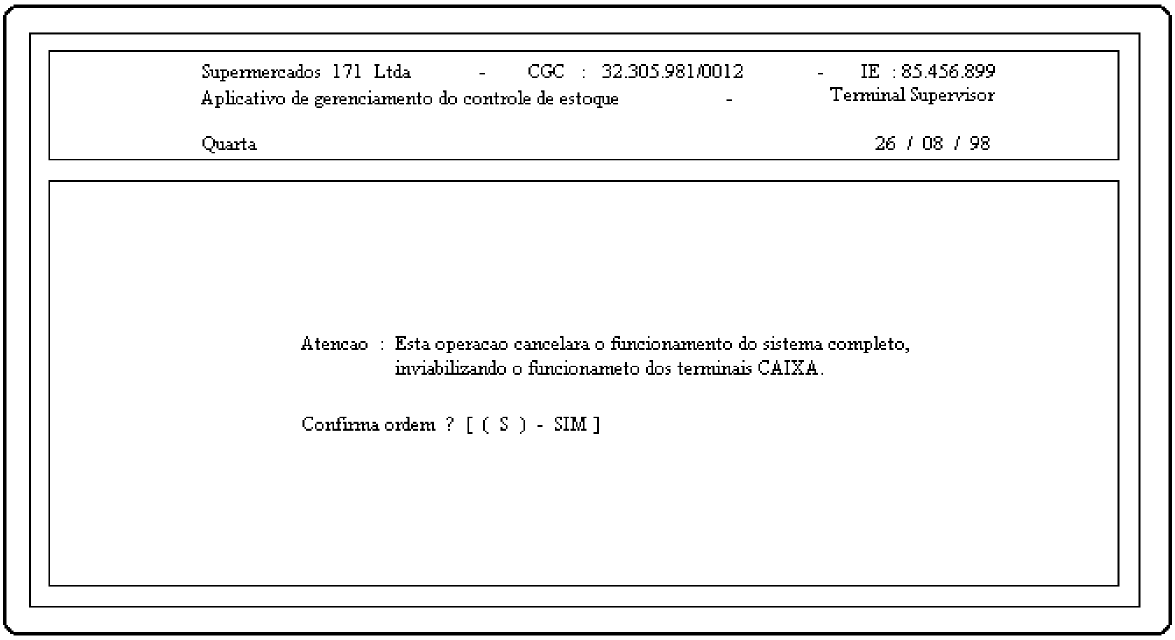
>> Deseja nova consulta ? [(S)im]

Tela 7 - Consulta individual a um produto



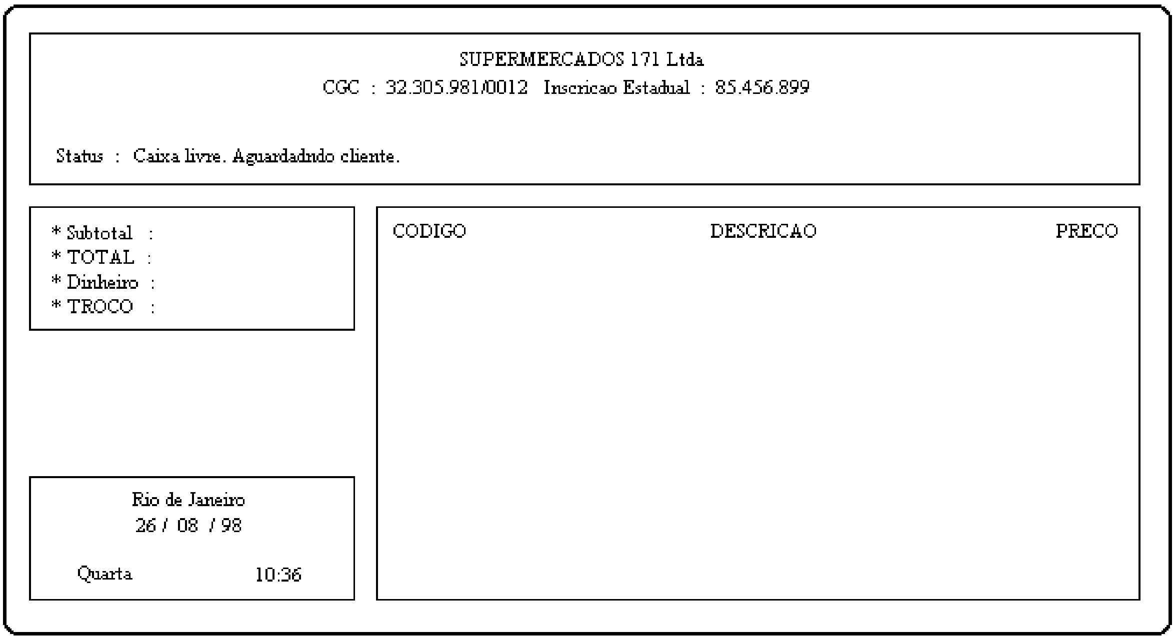
Tela 8 - Ordem de atualização para o PAI

Escolhendo a opção 8 o que se quer é encerrar o funcionamento do sistema. Se o GERENTE parar de executar, nenhuma operação no sistema poderá ser realizada. Por isso a mensagem de confirmação alerta o usuário. A tela 9 mostra esta situação.



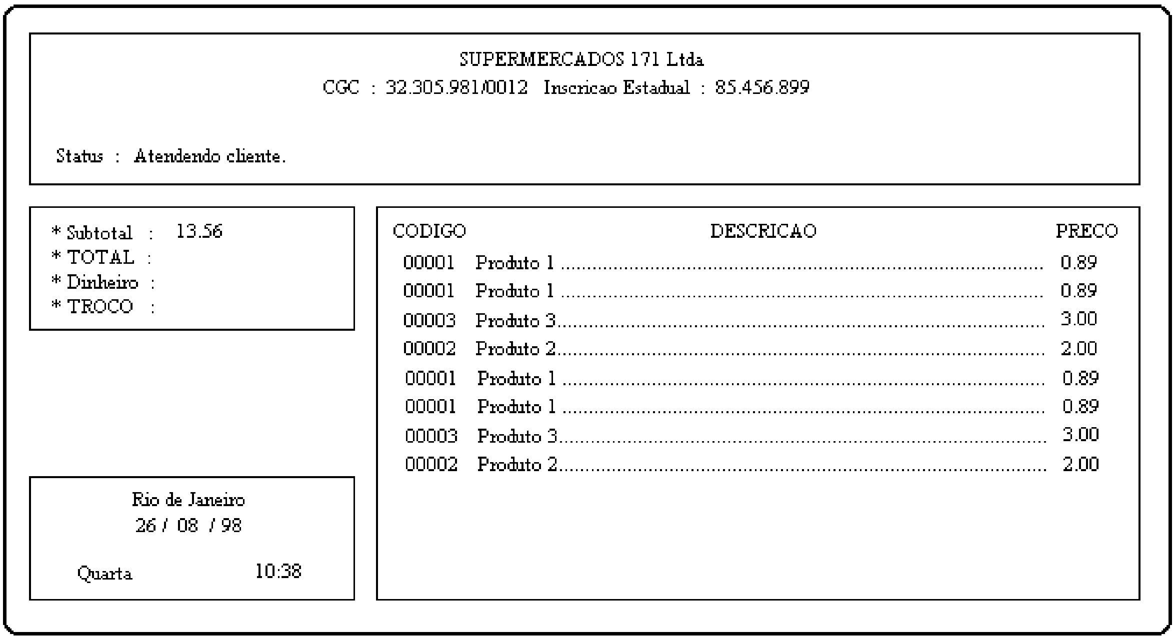
Tela 9 - Fim de operação do GERENTE

Para conectar um CAIXA ao sistema devemos entrar no diretório equivalente (para mais detalhes consulte o anexo C) e digitar "./caixa". Após as mensagens iniciais veremos a tela 10, caracterizando a espera por algum cliente.



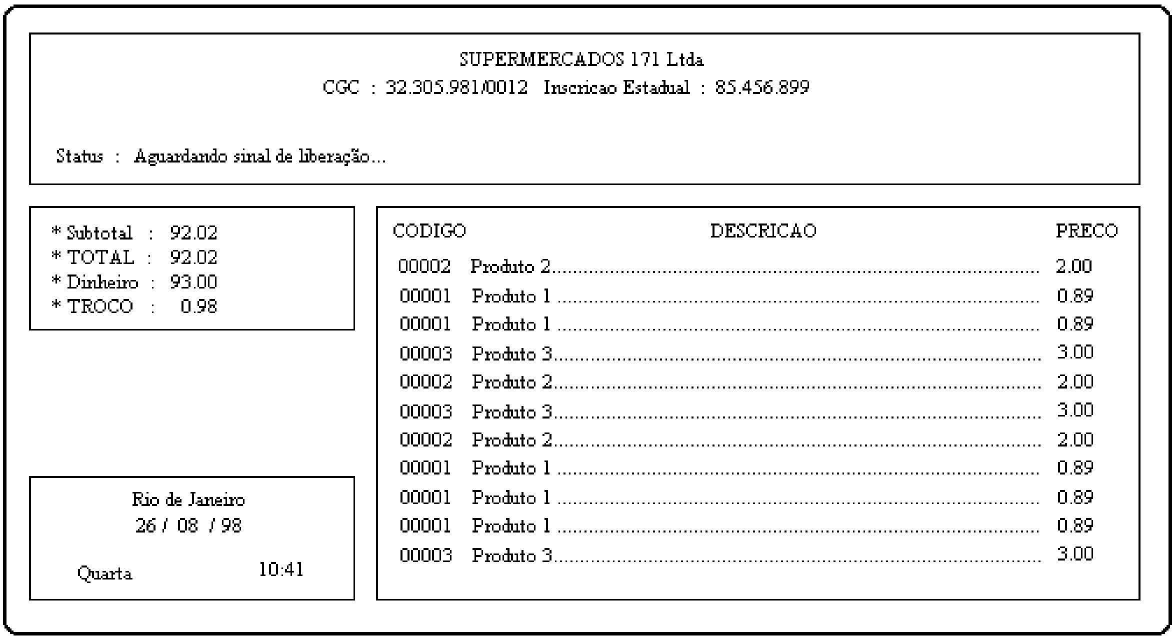
Tela 10 - CAIXA aguardando cliente

Assim que um cliente chega (lembre-se de que isto é uma simulação) os produtos começam a ser registrados. Após a passagem de alguns produtos o formato de tela é mostrado na tela 11.



Tela 11 - CAIXA atendendo cliente

Finalmente, após o atendimento do cliente, há o envio do relatório para o supervisor e o CAIXA entra em modo de espera de outro cliente. A tela 12 mostra esta situação.



Tela 12 - CAIXA aguardando sinal de liberação do SUPERVISOR

Basicamente estas são as telas principais presentes nos programas desenvolvidos. Algumas variações são possíveis. O importante é que as telas são auto explicativas, o que facilita a interação do usuário com o sistema.

4.10) - Comentários

Neste capítulo comentamos, de maneira global, cada um dos sete processos que foram implementados seguindo a modelagem por REDES DE PETRI desenvolvida nas seções anteriores. O principal enfoque foi dado à relação entre as técnicas utilizadas e os exemplos que foram desenvolvidos e comentados no capítulo 2 desta documentação. Todos os programas somam mais de três mil linhas de código. Obviamente não comentamos cada linha destes programas. Além dos comentários feitos neste capítulo, temos uma documentação contida nos próprios programas e que também é de grande utilidade.

Em particular podemos notar que os programas desenvolvidos na categoria EXEMPLOS foram intensamente utilizados na elaboração dos programas que implementam nossa solução. Pedaços de código foram diretamente copiados, enquanto alguns outros sofreram pequenas mudanças. Na verdade este era o nosso objetivo principal.

Sendo assim termina a nossa tarefa de resolução do problema hipotético. Na próxima seção iremos proceder nos comentários e conclusões finais, revisando o que foi feito ao longo de todas as fases anteriores.

Capítulo 5 - Comentários e conclusões finais

Este trabalho procurou apresentar o sistema operacional LINUX através da resolução de um problema característico utilizando o mesmo. Esta solução foi implementada apenas a nível de um "software" de simulação. Isto significa que foi suposto que a estrutura física e de "hardware" necessária já existia. O LINUX possui, como característica mais marcante, a capacidade de multiprocessamento de tarefas. Na verdade ele é um sistema operacional do tipo UNIX e que pode ser executado em computadores da família PC.

Para que este problema pudesse ser resolvido com sucesso seguiu-se algumas etapas. Inicialmente apresentamos os conceitos mais importantes a serem utilizados na nossa solução como, por exemplo, PROCESSOS, OPERAÇÃO COM PROCESSOS, COMUNICAÇÃO ENTRE PROCESSOS POR TROCA DE MENSAGENS UTILIZANDO SOCKETS e ACESSO DISCIPLINADO À REGIÕES COMPARTILHADAS DE DADOS COM EXCLUSÃO MÚTUA UTILIZANDO SEMÁFOROS. Para cada conceito abordado e que foi utilizado na nossa solução apresentamos e comentamos programas desenvolvidos em ANSI C. Estes programas também contêm documentação em forma de comentários no código, o que pode ser útil na sua análise.

Após esta apresentação inicial do sistema LINUX foi realizada, então, a descrição do problema. Esta descrição incluiu as exigências a serem cumpridas, tais como escalabilidade do número de terminais a serem conectados na rede e do tamanho da lista de produtos vendidos no supermercado.

Baseando-se na descrição do problema foi proposta uma solução dentro de um ambiente multitarefa. Para cada tipo de terminal presente na rede enumeramos as tarefas que deveriam ser realizadas e as dividimos entre processos de modo a proporcionar a maior eficiência possível. A partir destas atribuições modelamos cada um dos processos por uma REDE DE PETRI. Estas redes são uma ferramenta de modelagem e que podem ser definidas como uma extensão dos diagramas de estados. Conforme foi visto, foi possível dividir o sistema em duas partes principais. Uma delas cuidava da atualização dos preços dos produtos em tempo real enquanto a outra cuidava do controle de estoque em tempo real. Organizamos o sistema, então, em duas redes gerais. Para cada uma destas redes mostramos os processos envolvidos e os lugares comuns que representavam as mensagens que trafegavam na rede. Para a validação destes modelos utilizamos um "software" que foi desenvolvido especificamente para este propósito : o ARP (Analisador de Redes de Petri). Para que estas redes pudessem ser submetidas à análise criamos os arquivos que as descreviam no formato utilizado pelo programa. Estes arquivos foram apresentados e submetidos à análise do mesmo. O resultado da análise retirado do arquivo gerado pelo próprio analisador foi apresentado. Este resultado validou as redes analisadas, mostrando que o sistema não bloquearia e que não haveria sobrecarga de mensagens trafegando no meio físico. Na verdade, estas conclusões foram tiradas através da informação das propriedades atendidas por estas redes, que é fornecida pela análise do programa. Desta forma, nos baseando diretamente nas REDES DE PETRI validadas pelo ARP, desenvolvemos os programas que implementaram cada processo no sistema. O único processo que não foi modelado foi o GERENTE, já que se trata de uma aplicação clássica em banco de dados e que este não se comunica com qualquer outro processo através da troca de mensagens.

Foram comentados, então, os programas que implementam a nossa solução. Estes comentários foram globais. Entretanto alguns extratos (os mais importantes) do código foram retirados e comentados em particular. A execução final destes programas mostra o sucesso do processo de modelagem, já que não foram detectados problemas na execução do sistema.

A capacidade de multiprocessamento do LINUX foi amplamente utilizada. Conforme foi visto, no terminal SUPERVISOR teremos executando em paralelo, pelo menos, quatro processos que dizem respeito à nossa solução. Conforme as conexões forem sendo estabelecidas e novas cópias dos processos sejam criadas para atender estas conexões este número pode crescer. Por isso estudamos detalhadamente as técnicas para duplicação de processos, lançamento de outros processos e finalização de processos.

A comunicação entre processos também foi técnica fundamental para a realização de nosso sistema. A comunicação por troca de mensagens foi especialmente interessante. Utilizamos os SOCKETS, que são bastante gerais por admitirem comunicação entre processos remotos (ou seja, que executam em máquinas diferentes). O acesso disciplinado através da utilização dos SEMÁFOROS (que também são uma forma de comunicação entre processos) também foi essencial para a nossa implementação. Através desta facilidade foi possível garantir acesso com exclusão mútua aos arquivos compartilhados de dados, evitando possíveis problemas e perda de informações. Também foi utilizada uma terceira forma de comunicação entre processos : através de arquivos em disco. Os principais arquivos utilizados foram o "I.chr", "I2.chr". Estes arquivos serviram como aviso para que um outro processo tomasse determinado procedimento.

Este nosso projeto priorizou o aspecto de disponibilidade para alunos e pessoas interessadas. Por este fato tivemos preocupação com a portabilidade do sistema. Devido ao fato de que os programas estarão disponíveis para consulta e execução na máquina "MAGMA" do DEL da UFRJ, nos preocupamos com a portabilidade para o FreeBSD. As pequenas mudanças que tornam possível esta portabilidade estão comentadas de maneira apropriada no anexo B. Entretanto, vale a penas ressaltar que foram encontradas algumas dificuldades, já que o sistema foi desenvolvido originalmente no LINUX. A solução encontrada foi utilizar os comandos definidos no "ANSI C STANDARD". Por muitas vezes trechos foram desenvolvidos com sucesso no LINUX e não executaram no FreeBSD, obrigando-nos a sucessivas mudanças e ajustes. Deste maneira, aconselhamos o leitor que pretende desenvolver sistemas portáveis a consultar o manual e verificar se as funções que pretende utilizar são padrão para os dois ou mais sistemas. Também vale a pena lembrar que as principais incompatibilidades aparecem nas funções de manipulação de tela (interface). Portanto devemos redobrar atenção neste ponto.

Um problema interessante que foi enfrentado diz respeito ao atendimento da exigência de escalabilidade do número de caixas que poderiam se conectar ao sistema. Note que o ponto crítico diz respeito às atualizações de preços em tempo real. Quando o GERENTE autoriza a mudança de um preço ou da descrição de um item, esta mudança deve ser transmitida para todos os caixas em funcionamento. Logo, a informação de quantos caixas estão presentes na rede e a maneira com a qual podemos nos comunicar com eles deve estar presente no terminal SUPERVISOR. A solução adotada foi a criação de uma tabela contendo a identificação do caixa e a porta na qual podemos requisitar conexão para envio de mensagens. Logo, cada caixa possui uma porta proprietária e única e um número de identificação. O programa "config" permite que o usuário possa configurar estas informações. A necessidade da definição de uma porta lógica proprietária para cada CAIXA nasceu do fato de que estes programas poderão ser executados em uma mesma máquina. Conforme já foi explicado, caso cada CAIXA fosse executado em um terminal distinto, seria possível definir uma única porta lógica e requisitar as conexões utiliozando o IP de cada máquina. Quando algum caixa é desligado ele não envia a informação de que está fora da rede para o terminal SUPERVISOR. Para compensar isso adotamos um procedimento no qual o processo PAI tenta três vezes se comunicar com o caixa. Caso as três tentativas sejam fracassadas as informações relativas ao caixa em questão são invalidadas. Neste caso, o fracasso na tentativa de estabelecimento da conexão é entendido como desligamento do caixa. Isto nem sempre é verdade, pois podem ocorrer outras situações. Apesar disso, o fracasso de três tentativas pode significar que algo está errado como, por exemplo, problemas no meio físico. Para o nosso projeto a solução foi válida, embora não tecnicamente perfeita.

Uma outra dificuldade diz respeito ao tratamento de erros. Conforme foi visto, apenas um processo tinha acesso à interface com o usuário em cada terminal (CAIXA no terminal CAIXA e GERENTE no terminal SUPERVISOR). Então, caso ocorresse um erro em algum dos processos que não tivessem acesso à tela, como saberíamos que erro foi este? Para isto poderíamos ter utilizado duas técnicas : enviar a mensagem de erro através da comunicação dos processos em BACKGROUND com o processo em FOREGROUND ou armazenar a informação do erro em alguma localidade segura. Escolhemos a segunda opção por questões de simplicidade. Sendo assim, quando ocorre algum erro que impossibilita a execução, a informação é gravada em um arquivo de disco. Esta técnica foi utilizada nos processos PTA, SUPERVISOR, enviaTP e PAI. Caso o sistema não funcione apropriadamente estes arquivos podem ser consultados, o que auxiliará na manutenção dos programas.

Um outro problema foi detectado quando já pensávamos que o código estivesse finalizado. O sistema funcionava normalmente. Para cada duplicação dos processos para atendimento de conexões, havia um aumento do número de processos executando no sistema. O problema se localizou no momento em que as cópias terminavam a execução. Estes processos finalizados continuam constando na tabela de processos sob o estado de "<zombie>". Isto significa que o processo já não executa mais e que as suas áreas necessitam ser liberadas. Como o processo que lançou estas cópias continua em execução torna-se necessário a execução da primitiva "wait", que liberará as áreas destinadas aos processos cópia. Caso não se libere estes processos a tabela crescerá, alcançará o limite máximo e o sistema não funcionará adequadamente. O problema é que esta primitiva trava o processo que faz a chamada até que uma cópia encerre a execução. Logo, seria necessária a informação de quantos processos cópia (dos que foram lançados) já encerraram a execução, de modo que a execução do "wait" não travaria o processo original. Sendo assim optamos por criar um arquivo que será atualizado pelas cópias antes da finalização das mesmas. Esta atualização consiste em um acesso disciplinado (para isso criamos um semáforo apenas para este objetivo) no qual a cópia lê o número contido no arquivo e grava este valor acrescido do valor 1. Quando o número de cópias lançadas alcança 10 (número arbitrário), o processo original acessa este arquivo e procede em um "loop" executando a primitiva "wait" para enterrar os processos zumbis. A parte do código que realiza isto é facilmente reconhecível, além de estar comentada. Esta técnica é utilizada pelos processos enviaTP e SUPERVISOR. É bom lembrar que nos nossos comentários sobre estes processos não nos referimos a estas partes do código.

O aspecto de didática deste projeto foi priorizado. Os programas finais desenvolvidos e que estão sendo colocados à disposição dos interessados procuraram seguir as boas técnicas de programação. Muitas mudanças poderiam ser feitas caso se quisesse caracterizar este projeto como uma solução real e passível de implementação. Entretanto acreditamos ter atendido à proposta inicial do projeto.

Com certeza este trabalho pode ter desdobramentos interessantes. Não acreditamos que a elaboração de outras versões dos programas da nossa solução seja aconselhável. Apesar de estarem satisfatoriamente documentados, o entendimento poderia demandar muito tempo. O que poderia ser feito é uma continuação do nosso trabalho dentro do mesmo propósito. Isto envolveria a abordagem de novos conceitos nos sistemas operacionais e até mesmo a utilização de outros sistemas que não o LINUX. Neste caso poderia ser realizado um novo estudo de caso, abordando os conceitos utilizados neste projeto e os demais abordados neste eventual novo projeto. A mudança da interface gráfica para um ambiente de X-WINDOWS é fortemente recomendável. Propostas para um novo estudo de caso poderiam envolver a implementação de um BANCO VIRTUAL acessível pela INTERNET em um ambiente multitarefa, um programa de projeto de circuitos eletrônicos em que várias pessoas pudessem interagir em tempo real através de terminais remotos conectados à INTERNET, de modo que cada mudança sugerida por um dos participantes pudesse ser transmitida a todos os outros antes de aprovada. Também neste caso um ambiente multitarefa pode ser vantajoso.

Os estudos de caso que podem ser realizados são em grande número. Apesar disto, uma continuação deste projeto poderia ser somente a realização de uma solução de um problema real a ser, de fato, implementada. Isto poderia ser interessante pois o aluno deveria priorizar outros aspectos que não foram relevantes neste nosso trabalho como custo, velocidade e eficiência do sistema. Neste caso as duas soluções poderiam ser comparadas no que diz respeito a estes parâmetros. Ou seja, os conceitos sobre sistemas operacionais continuariam a ser utilizados, só que agora sob prisma de análise diferente. Neste caso o aluno chegaria à conclusão de que outras técnicas seriam mais vantajosas dependendo da aplicação (por exemplo, os *"threads"* seriam melhor opção do que a duplicação de processos em termos de eficiência). Na verdade, qualquer trabalho que utilize um sistema operacional multitarefa terá alguma relação com o presente.

Uma última proposta de continuidade poderia envolver aplicações sobre sistemas distribuídos. Este é um assunto bastante atual sobre o qual se pode encontrar vasta documentação. Isto facilitaria a elaboração de futuros trabalhos.

Bibliografia

H. Schildt - ***C : The Complete Reference***, Berkley, McGraw-Hill, 1990;

H. Schildt - ***Linguagem C : Guia do usuário***, São Paulo, McGraw-Hill, 1986;

A. S. Tanembaum - ***Modern Operating Systems***, New Jersey, Prentice Hall, 1992;

E.P. Silva e R.P.Silva - ***Projeto Didático de Protocolo da Camada de Rede***, Rio de Janeiro, DEL-UFRJ, 1997;

P. Volkering, K. Reichard e E. Foster-Johson - ***Linux : Installing and Configuring,*** New York, Mis:Press, 1997;

H.M.Deitel - ***Operating Systems***, Boston College, Addison-Wesley Publishing Company, 1990;

James L. Peterson **- *Petri Net Theory and the Modeling of Systems***, New Jersey, Prentice Hall, 1981;

W. Richard Stevens - ***Unix Network Programming***, New Jersey, Prentice Hall, 1990;

Anexo A : Comentários acerca da instalação do LINUX

O LINUX (pronuncia-se *Lai-nucs*) é um sistema operacional multitarefa de 32-bits desenvolvido para ser utilizado em PCs INTEL 386 (ou superiores). Tecnicamente o Linux é um sistema do tipo UNIX, o que significa que responde aos comandos padrão UNIX e executará programas UNIX.

O Linux nasceu de um projeto desenvolvido por uma única pessoa, Linus Torvalds (então um estudante na Universidade da Finlândia em Helsinki), que desejava sua própria alternativa para outra alternativa de UNIX que havia sido desenvolvida (Minix, implementada pelo conhecido Andrew S. Tanenbaum). Ele projetou o Linux para ser similar ao Minix, embora mais estável e de distribuição livre.

Por muito tempo o Linux foi um sistema operacional em desenvolvimento, de modo que muitas versões "beta" circularam no mundo virtual (principalmente na Internet). A versão 0.2 foi liberada na metade de 1991; em 1994 a versão 1.0 foi finalmente liberada. Um grupo de voluntários ajudou Torvalds a terminar o Linux. A partir de então foram desenvolvidas outras versões que melhoraram a qualidade do sistema em alguns aspectos como, por exemplo, facilidade de instalação. O Linux é um sistema totalmente aberto. Você tem acesso ao código fonte e, se quiser fazer alterações no sistema, pode fazê-las. Esta é uma característica bastante interessante.

A exemplo de todos os sistemas do tipo UNIX para PCs (inclusive SCO UNIX e FreeBSD), o Linux é bastante meticuloso acerca do "hardware" em que está executando. Queremos dizer que ele realiza muitas interações diretas com o "hardware" do PC. O DOS, em comparação, é um sistema muito menos rigoroso, e portanto, mais capaz de executar em quase qualquer PC. Ele ignora algumas falhas na arquitetura PC.

O Linux, entretanto, exporá estas falhas. Como um UNIX para PC, o Linux interage muito próximo ao "hardware" do PC, escrevendo diretamente nos seus vários componentes. Se houver algum problema com o seu PC (mesmo que pequeno), ele será encontrado pelo Linux. Nós assumimos que o leitor tem algum conhecimento sobre seu PC e necessita de uma formalização acerca de seus componentes. Vamos fornecer, então, a configuração mínima e a ideal para que se possa executar o Linux aproveitando todo o seu potencial. Estas informações estão contidas na tabela 1 a seguir :

Componente do PC	*Mínimo*	*Ideal*
Processador	Intel 80386 ou equivalente	Pentium (acima de 100 Mhz)
Memória RAM	4Mb (8Mb para rodar X-Windows)	16Mb
Placa de Vídeo	Padrão VGA	Padrão SVGA
Porção livre no HD	125Mb	500Mb ou mais
Barramento	Qualquer um menos MCA	Qualquer um menos MCA
CD-ROM	Dupla velocidade	Quádrupla velocidade
Mouse	Logitech ou compatível	Logitech ou compatível
Placa de rede	Nenhuma	Admite (se ligado em rede)

Outros componentes como impressoras, placas de som, joystick, scanners podem ser reconhecidos e funcionar sobre o Linux. Como pode se ver, a configuração mínima para que se possa executar o Linux em seu computador pessoal é bastante acessível. Com o ritmo frenético da tecnologia, podemos afirmar que a configuração ideal para executar o Linux já é razoavelmente ultrapassada, de modo que não acreditamos haver problema, sob a aspecto de requisitos básicos, para a instalação. Alguns sistemas particulares podem apresentar, em situações particulares, problemas durante a execução do Linux (principalmente nas máquinas mais antigas, como 486 e 386 com processadores de outros fabricantes que não a INTEL). Mais detalhes podem ser encontrados nas referências bibliográficas fornecidas no fim da documentação.

Existem distribuidores diferentes do Linux. Basicamente os mais populares são a versão “Red-Hat” e “Slackware”. As diferenças entre eles se situam, principalmente, na fase de instalação. O KERNEL deles é o mesmo. Pela vivência do autor o “Red-Hat” apresenta instalação mais simples e, pelo menos na versão que foi utilizada para o desenvolvimento do projeto, não apresentou problemas no reconhecimento do CD-ROM (problema que foi vivenciado quando da tentativa de instalação do Slackware, tornando-a, inclusive, inviável). As versões mais atuais (em média de seis em seis meses temos atualizações disponíveis) talvez tenham eliminado este incômodo problema. Apesar disso é bom ressaltar que o CD-ROM utilizado foi um de sêxtupla velocidade fabricado pela CREATIVE e que não é muito comum. Desconsiderando este problema na instalação não temos motivos para aconselhar o leitor a utilizar uma ou outra versão (como se vê não queremos nos comprometer…).

Em geral os folhetos que acompanham os CDs de instalação são esclarecedores e suficientes para que se possa proceder na instalação de maneira correta. Se o leitor achar necessário pode consultar as referências bibliográficas fornecidas no fim da documentação e que podem esclarecer algumas dúvidas e auxiliá-lo. Antes de terminar este anexo vamos fazer uma observação importante, e que, se tivesse sido lidas por nós antes, teria facilitado muito a instalação do Linux.

Esta observação diz respeito ao velho problema do CD-ROM. Se na instalação o seu CD-ROM não consegue ser detectado (o que impossibilita a instalação já que o sistema está no CD-ROM), verifique se ele está ligado diretamente na controladora da placa mãe. Alguns CD-ROMS antigos funcionavam ligados à placa de som, e isto geralmente impossibilita o seu reconhecimento. Verifique também o nome do fabricante e o modelo coreto na documentação do aparelho, para que estes dados possam ser passados corretamente na hora da instalação.

Esperamos que este guia rápido possa auxiliar o leitor na instalação do LINUX.

Anexo B : Comentários acerca da portabilidade

O nosso projeto foi desenvolvido no sistema operacional LINUX, em um computador pessoal PC 486 INTEL. Entretanto, havia a exigência de que os programas pudessem ser executados, com o mínimo de mudanças possíveis, na máquina MAGMA presente no departamento de eletrônica da UFRJ. Nesta rede temos um sistema operacional que também é do tipo UNIX para PCs : o FreeBSD. O Linux e o Free BSD possuem algumas diferenças e causam muitas discussões entre os usuários que se identificam com cada uma deles. Discussões à parte, fizemos os ajustes necessários para que os alunos pudessem executar os programas deste projeto em qualquer do dois sistemas sem muita complicação. Entretanto algumas diferenças básicas, como argumentos passados para compilação, não puderam ser compatibilizadas, por serem características intrínsecas dos sistemas. Na tabela 1 temos as linhas de comando para a compilação de cada programa desenvolvido tanto para o FreeBSD como para o Linux. Vejamos :

Programa	***FreeBSD***	***Linux***
duplica.c	gcc -o duplica duplica.c -Wall	gcc -o duplica duplica.c -Wall
triplica.c	gcc -o triplica triplica.c -Wall	gcc -o triplica triplica.c -Wall
pai.c	gcc -o pai pai.c -Wall	gcc -o pai pai.c -Wall
filho.c	gcc -o filho filho.c -Wall	gcc -o filho filho.c -Wall
finaliza.c	gcc -o finaliza finaliza.c -Wall	gcc -o finaliza finaliza.c -Wall
sem1.c	gcc -o sem1 sem1.c -Wall	gcc -o sem1 sem1.c -Wall
cliente.c	gcc -o cliente cliente.c -Wall	gcc -o cliente cliente.c -Wall
servidor.c	gcc -o servidor servidor.c -Wall	gcc -o servidor servidor.c -Wall
caixa.c	gcc -lncurses -o caixa caixa.c -Wall	gcc -lncurses -o caixa caixa.c -Wall
pta.c	gcc -o pta pta.c -Wall	gcc -o pta pta.c -Wall
config.c	gcc -o config config.c -Wall	gcc -o config config.c -Wall
gerente.c	gcc -lncurses -o gerente gerente.c -Wall	gcc -lncurses -lndbm -o gerente gerente.c-Wall
pai.c	gcc -o pai pai.c -Wall	gcc -lndbm -o pai pai.c -Wall
enviaTP.c	gcc -o enviaTP enviaTP.c	gcc -lndbm -o enviaTP enviaTP.c -Wall
supervisor.c	gcc -o supervisor supervisor.c -Wall	gcc -lndbm -o supervisor supervisor.c -Wall
par.c	gcc -o par par.c -Wall	gcc -o par par.c -Wall

Tabela 1 - Comparação entre modo de compilação no Linux e FreeBSD

Se tudo ocorrer de forma correta, não deve ser vista nenhuma mensagem durante as compilações requisitadas através dos pedidos descritos. Para cada programa fonte teremos, então, um executável com o mesmo nome excetuando-se o sufixo ".c".

Outra mudança principal é a mudança do "host name", que é característica de cada máquina. Para a máquina MAGMA o arquivo "hostname.h" já contêm o dado correto "magma.del.ufrj.br". Para executar os programas de nossa solução no sistema LINUX caseiro (admitindo PC não conectado em rede), este conteúdo deve ser alterado para "localhost". Sem este dado é impossível estabelecer as conexões entre os processos.

Com intuito de facilitar ao máximo a utilização dos programas disponibilizados, foram criados diversos "makefiles" (arquivos que realizam a compilação dos programas automaticamente) que estão localizados dentro dos diretórios criados. Abaixo vamos descrever as linhas de comando necessárias para que estes "makefiles" realizem as compilações e façam, automaticamente, a mudança no arquivo "hostname.h", quando necessário. Desta forma, não há a necessidade de se realizar nenhuma mudança manual por parte do usuário para que se possa executar estes programas no LINUX.

Diretório	***Linha de comando***
automacao/exemplos/lancamento_processos	make
automacao/exemplos/semaforos	make
automacao/exemplos/sockets	make
automacao/solucao/caixa	make caixa
automacao/solucao/supervisor	make
automacao/solucao/deposito	make deposito

O formato dos arquivos utilizados na nossa solução (TP.db, ESTOQUE.db, I.chr, I2.chr, rede, porta, COMPRAS, listprod) não é portável. Isto significa que se você copiar estes arquivos e tentar executar os programas no Linux, haverá problemas de compatibilidade, e a execução não será possível. Quando for executar os programas pela primeira vez no Linux, execute o GERENTE e crie sua própria lista de produtos. Os demais arquivos serão criados pelos próprios programas, não havendo necessidade de se preocupar.

Estas são as mudanças necessárias para executar os programas que serão copiados da máquina "MAGMA" para seu sistema "Linux". Como se vê o conteúdo de nenhum programa deve ser alterado, mostrando que a proposta de portabilidade foi alcançada com sucesso.

Anexo C : Manual do usuário

Organizamos os programas que foram desenvolvidos em subdiretórios distintos. Melhor do que descrever é visualizar a organização feita através da figura C.1 abaixo :

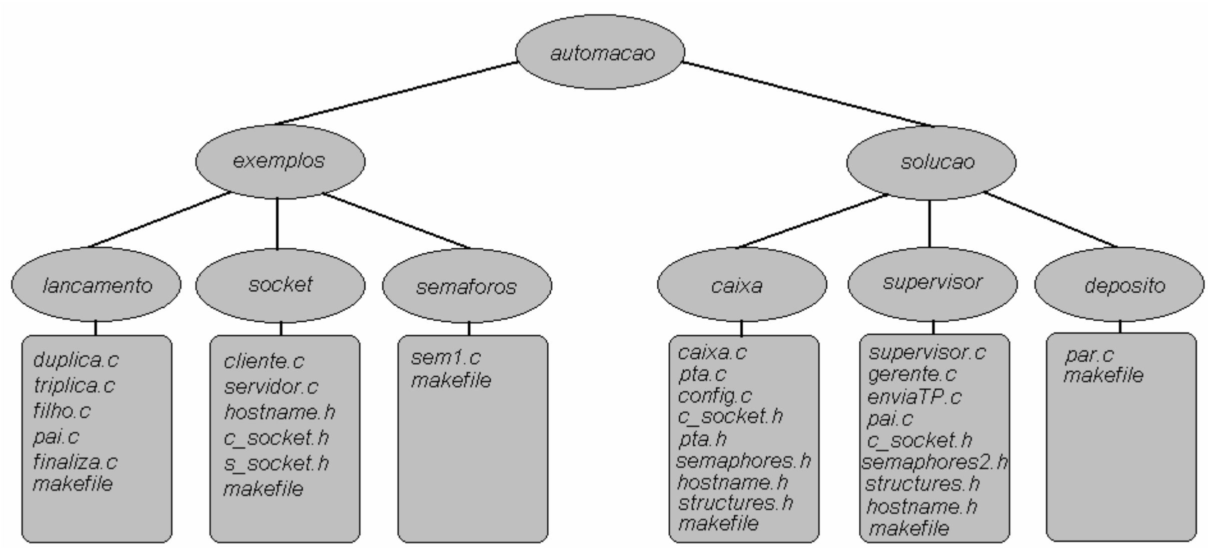


Figura C.1 - Organização dos programas desenvolvidos em diretórios

Como podemos notar, inicialmente, o aluno tem acesso aos programas fontes em ANSI C. Para que se possa executá-los é necessário a compilação destes programas da maneira que é descrito no ANEXO B. Como sabemos, a execução de algum programa é feita se digitando "./" e o nome do programa.

Os programas da categoria EXEMPLOS que devem ser executados são : "./duplica", "./triplica", "./filho", "./pai", "./finaliza", "./sem1", "./servidor &" (servidor em BACKGROUND) e "./cliente".

No caso do último exemplo, caso se queira conectar mais de um cliente, é necessário abrir novas seções. Acessando a máquina MAGMA isto pode ser feito através de "ALT + A" e "logando-se" novamente. Para mudar de uma seção aberta para outra devemos teclar "ALT + N". No Linux, utilizamos "ALT + tecla de função relativa à seção que se quer acessar". Por exemplo, "ALT + F2" acessa a segunda seção. O Linux permite a abertura de seis seções diferentes.

Os programas da categoria ESTUDO DE CASO que precisam ser executados são os seguintes: "./gerente", "./par" e "./caixa". Na verdade a idéia inicial seria executar cada um destes programas em um terminal diferente. Entretanto, a situação mais comum será a execução em um mesmo terminal, por questões de praticidade. Para que isso seja possível devemos abrir, então, três ou mais seções. Se o leitor desejar executar os programas em terminais diferentes pode fazê-lo sem maiores problemas logando-se em várias máquinas diferentes conectadas à rede.

O GERENTE executa em uma seção particular. Aconselha-se executá-lo na primeira seção. Este programa deve ser executado em primeiro lugar, para que se possa conectar CAIXAS à rede. Na primeira execução é necessário criar uma lista de produtos, que pode ser feito através da primeira opção do "menu". Digite quantos produtos quiser.

O PAR deve ser executado a seguir em uma seção separada (segunda, por exemplo).

A partir daí devemos organizar diretórios com a denominação "caixa1", "caixa2", etc., onde cada diretório simulará um terminal CAIXA que deverá conter os seguintes programas : "caixa.c", "pta.c" e "config.c". Dentro destes diretórios devemos executar, inicialmente, o "./config". Este é o programa para configuração da porta de comunicação. Responda as perguntas atribuindo a identificação do CAIXA e a porta. A escolha da porta pode ser aleatória dentro do intervalo permitido. Entretanto, não se pode atribuir uma mesma porta para dois CAIXAS. O sistema não têm como verificar esta situação. Se você atribuir a mesma porta, a execução produzirá resultados não desejados, podendo até ser interrompida. Após a execução deste aplicativo, cada seção representará um terminal caixa. Para conectar um caixa, entre no diretório respectivo e digite "./caixa". Todos os outros arquivos serão criados automaticamente, e o PTA será lançado automaticamente.

Para visualizar os resultados é só selecionar cada seção e acompanhar a simulação. Pode-se consultar as informações acerca de um produto e verificar que as baixas estão sendo efetuadas conforme os clientes estão sendo atendidos. Além disso, quando o estoque interno ficar baixo podemos ver o pedido de autorização que será mostrado no terminal do depósito.

Tanto os programas exemplo quanto a simulação são auto-explicativos. Siga os passos acima, execute os programas da maneira descrita e acompanhe a execução.

Anexo D : Listagem dos programas da categoria de EXEMPLOS

D1 – *duplica.c*

```c
#include <stdio.h>
#include <stdlib.h>  // system()
#include <unistd.h>  // fork(), getpid(), sleep()

void main(void)
{
  // declaracao de variaveis locais
  int pid, contador, pid_original;

  // atribui o 'pid' do processo corrente a variavel pid_pai
  pid_original = getpid();

  system("clear");
  printf("%58s\n","Exemplo 1 de lancamento de processos");
  printf("%58s\n","-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_");
  printf("Duplicando processo ORIGINAL... \n");
  sleep(2);

  //A linha abaixo duplica o processo corrente
  pid = fork();

  if (pid == 0) // copia executando
  {
    sleep(1);
    for ( contador=1 ; contador <= 3 ; contador++)
    {
      printf("%49s\n","Processo COPIA executando");
      sleep(2);
    }
    sleep(1);
  }

  else if (pid > 0) // ORIGINAL executando (retorna o pid do processo duplicado)
  {
    printf("ORIGINAL   tem PID = %d\n",pid_original);
    printf("COPIA      tem PID = %d\n\n",pid);
    for ( contador=1 ; contador <= 3 ; contador++)
    {
      printf("Processo ORIGINAL executando\n");
      sleep(2);
    }
    printf("\nVerifique o PID da COPIA na tabela abaixo:\n");

    // a linha abaixo mostra os processos que estao sendo executados
    system("ps"); // "system call" que mostra os processos correntes
  }

  // se o programa executar a linha abaixo houve algum erro na duplicacao
  else printf("Houve algum erro na duplicacao do processo ORIGINAL...\n");
}
```

D2 – *triplica.c*

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
// funcao que imprime 3 mensagens de identificacao na tela
void loop_copia(num_copia)
{
  // declaracao de variaveis locais
  int contador;
  for ( contador=1 ; contador <= 3 ; contador++)
  {
       printf("Processo COPIA %d executando\n",num_copia);
       sleep(2);
  }
  sleep(2);
}

void main(void)
{
  // declaracao de variaveis locais
  int pid , pid_original ;

  pid_original = getpid();

  system("clear");
  printf("%58s\n","Exemplo 2 de lancamento de processos");
  printf("%58s\n","-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_");
  printf("Triplicando processo ORIGINAL... \n");
  printf("ORIGINAL       tem PID = %d\n",pid_original);
  sleep(2);

  // a linha abaixo duplica pela primeira vez o processo corrente
  pid = fork();

  if (pid == 0) // primeira copia executando
  {
    loop_copia(1);
    exit(1); // primeira copia termina execucao
  }

  else if (pid > 0)         // PAI executando
  printf("Primeira COPIA tem PID = %d\n",pid);

  else printf("Houve algum erro na primeira duplicacao!!!");

  // a linha abaixo duplica pela segunda vez o processo corrente
  pid = fork();

  if (pid == 0) // segunda copia executando
  {
    loop_copia(2);
    exit(1); // segunda copia termina execucao
  }

  else if (pid > 0)          // PAI executando
  printf("Segunda  COPIA tem PID = %d\n\n",pid);

  else printf("Hove algum erro na segunda duplicacao!!!\n");

  // parte final da execucao do PAI
  sleep(6);
  printf("O processo ORIGINAL terminou a sua execucao.\n");
  printf("Verifique os PID's na tabela abaixo:\n");

  system("ps"); // "system call" que mostra os processos correntes
}
```

D3 – *pai.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define PROCESSO_FILHO "./filho"  // nome do FILHO que sera executado
```

```
void main(void)
{
  // declaracao de variaveis locais
  int pid, contador;

  system("clear");
  printf("%58s\n","Exemplo 3 de lancamento de processos");
  printf("%58s\n","-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_");
  printf("Duplicando processo PAI... \n");
  sleep(4);

  //A linha abaixo duplica o processo corrente
  pid = fork();

  if (pid == 0) // copia executando
  // as linhas abaixo fazem com que a copia seja substituida por ./filho
  {
    printf("Lancando processo ./filho & ... \n\n");
    execl (PROCESSO_FILHO, PROCESSO_FILHO, "&", NULL);
    _exit (EXIT_FAILURE); // termina execucao se houver algum erro
  }

  else if (pid > 0) // processo PAI executando
  {
    printf("PAI   tem PID = %d\n", getpid());
    printf("FILHO tem PID = %d\n",pid);
    sleep(1);
    for ( contador=1 ; contador <= 3 ; contador++)
    {
      printf("Processo PAI executando\n");
      sleep(2);
    }
    printf("Verifique o PID do FILHO na tabela abaixo:\n");
    system("ps"); // mostra os processos correntes
  }

  else printf("Houve algum erro na duplicacao do processo ORIGINAL...\n");
}
```

D4 – *filho.c*

```
#include <stdio.h>
#include <unistd.h>

void main(void)
{
  int contador;
  for ( contador=1 ; contador <= 3 ; contador++)
  {
    printf("%49s\n","Processo FILHO executando...");
    sleep(2);
  }
  sleep(2);
}
```

D5 – *finaliza.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void main(void)
{
  // declaracao de variaveis locais
  int pid;
```

```
  system("clear");
  printf("%58s\n","Exemplo 4 de lancamento de processos");
  printf("%58s\n","-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_");
  printf("Duplicando processo ORIGINAL... \n");
  sleep(4);

  //A linha abaixo duplica o processo corrente
  pid = fork();

  if (pid == 0) // copia executando
  {
    printf("Processo COPIA em loop infinito...\n\n");
    for(;;) // loop infinito
    {
    }
  }

  else if (pid > 0) // processo ORIGINAL executando
  {
    printf("ORIGINAL   tem PID = %d\n", getpid());
    printf("COPIA      tem PID = %d\n", pid);
    sleep(3);
    printf("Confira os PID's fornecidos na tabela abaixo :\n");
    sleep(3);
    system("ps");  // mostra tabela de processos correntes
    sleep(3);
    printf("\nAgora o processo ORIGINAL vai finalizar o processo COPIA...\n");
    sleep(2);
    kill(pid,SIGKILL);  // finaliza o processo COPIA
    printf("Confira agora a tabela de processos correntes : \n");
    sleep(3);
    system("ps");  // mostra a tabela de processos correntes
    sleep(3);
    printf("Processo ORIGINAL se auto-finalizando...\n");
    sleep(3);
    exit(0);  // encerra execucao
  }

  else printf("Houve algum erro na duplicacao do processo ORIGINAL...\n");
}
```

D6 – *sem1.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

extern int errno;

#define SEMPERM 0600

// a funcao abaixo cria o semaforo ou retorna o descritor se ja existe
// caso o semaforo esteja sendo criado ele e inicializado com valor 1
int initsem(key_t semkey)
{
  int status = 0 , id_sem ;

  if ((id_sem = semget( semkey, 1 , SEMPERM|IPC_CREAT|IPC_EXCL)) == -1)
  {
       if(errno == EEXIST)
       id_sem = semget(semkey,1,0);
  }
  else  // se ja existe...
```

```
       status = semctl(id_sem,0,SETVAL,1); // inicializa com valor 1

  if (id_sem == -1 || status == -1)
  {
       perror("Falha na inicializacao do semaforo");
       return(-1);
  }
  else
       return id_sem;  // tudo correto, retorna a identificacao do semaforo
}

// implementacao da primitiva DOWN()
int down(int id_sem)
{
  struct sembuf p_buf;
  p_buf.sem_num = 0;
  p_buf.sem_op  = -1;
  p_buf.sem_flg = SEM_UNDO;

  if(semop(id_sem, &p_buf, 1) == -1)
  {
       perror("Falha no wait()");
       exit(1);
  }
  else
       return(0);
}

// implementacao da primitiva UP()
int up(int id_sem)
{
  struct sembuf v_buf;
  v_buf.sem_num = 0;
  v_buf.sem_op  = 1;
  v_buf.sem_flg = SEM_UNDO;

  if(semop(id_sem, &v_buf, 1) == -1)
  {
       perror("Falha no signal()");
       exit(1);
  }
  else
       return(0);
}

// funcao de acesso a regiao critica
int acesso_disciplinado(key_t skey)
{
  int id_sem , pid = getpid();

  if ((id_sem = initsem(skey)) < 0)
       exit(1);

  printf("\n processo %d antes da regiao critica\n" , pid);

  down(id_sem); // aguarda no semaforo

  printf("\n processo %d na regiao critica\n" , pid);

  sleep(10); // neste ponto se faz o acesso a alguma regiao ou arquivo comum

  printf("\n processo %d saindo da regiao critica\n" , pid);

  up(id_sem); // libera semaforo

  printf("\n processo %d finalizando\n" , pid);

  exit(0); // fim da execucao
}

void main(void)
{
```

```
  key_t semkey = 0x200;

  system("clear");

  if (fork() == 0) // primeira copia do processo corrente executando
       acesso_disciplinado(semkey); // acesso disciplinado a regiao comum

  if (fork() == 0) // segunda copia executando
       acesso_disciplinado(semkey);

  if (fork() == 0) // terceira copia executando
       acesso_disciplinado(semkey);

  // aguarda que todas as copias terminem suas execucoes(tempo calculado)
  sleep(32);
}
```

D7 – *c_socket.h*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "hostname.h"

/****************************************************************************
*                                                                          *
*    A funcao abaixo solicita conexao com o SERVIDOR e retorna o descritor    *
*                a ser utilizado na comunicacao dos processos              *
*                                                                          *
****************************************************************************/
int open_client_socket(int clientport)
{
  int sockfd;
  struct sockaddr_in serv_name;
  struct hostent *hostPtr = NULL;

  bzero((char *) &serv_name, sizeof(serv_name));
  serv_name.sin_family      = AF_INET;
  serv_name.sin_port        = htons(clientport);

  hostPtr = gethostbyname(SERV_HOST_NAME);
  (void) memcpy(&serv_name.sin_addr,hostPtr -> h_addr,hostPtr -> h_length);

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
       return(0);
  if (connect(sockfd, (struct sockaddr *) &serv_name, sizeof(serv_name)) < 0)
        return(0);

  return(sockfd);
}
```

D8 – *s_socket.h*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <netdb.h>

/*****************************************************************************
*                                                                           *
*   A funcao abaixo cria um SOCKET para comunicacao com possiveis clientes    *
* Cada vez que um CLIENTE requisita conexao, o processo SERVIDOR eh duplicado *
*   e a copia atende ao CLIENTE, enquanto o processo original aguarda novas   *
*                             requisicoes para conexao.                     *
*                                                                           *
*****************************************************************************/

int open_server_socket(int serverport)
{
  int sockfd, newsockfd, clilen, childpid, on =1;
  struct sockaddr_in cli_addr, serv_addr;
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
       perror("server: can't open stream socket");

  if(setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,(const char *) &on,
                                                         sizeof(on)) < 0)
  perror("server: setsockopt error.");

  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port        = htons(serverport);

  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
       perror("server:can't bind local address");

  listen(sockfd,5);

  for(;;)
  {
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd , (struct sockaddr *) &cli_addr, &clilen);

    if (newsockfd < 0)
       perror("server : accept error");

    if ( (childpid = fork()) < 0)
       perror("server : fork error");

    else if (childpid == 0)
    {
      close(sockfd);
      return(newsockfd);
    }
    close(newsockfd);
  }
}
```

D9 – *cliente.c*

```
#include <ctype.h>
#include "c_socket.h"

// retorna um caracter lido e limpa o buffer
char le_caracter(void)
{
  // declaracao de variaveis locais
  char c;
  char temp;

  c = getchar();
  if (c != '\n')
  {
    do
```

```
    {
      temp = getchar();
    } while (temp != '\n');
  }
  return(c);
}

// prepara a mensagem a ser enviada com base nas informacoes do usuario
void prepara_mensagem(char *end_mens)
{
  // declaracao de variaveis locais
  char dados[256] , mensagem[107] , inicial[3];
  int chr_dados;

  do
  {
    system("clear");
    printf("%65s\n","Exemplo 1 de comunicacao entre processos por SOCKET");
    printf("%65s\n","_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_");
    printf("\n\nDigite uma sequencia de ate 100 caracteres :\n>");
    dados[0] = '\0';
    fgets(dados,256,stdin);
    dados[strlen(dados)-1] = '\0';
    chr_dados = strlen(dados);
    if ( chr_dados > 100 )
    {
       printf("\n Sequencia excedeu 100 caracteres\n");
       sleep(2);
    }
  } while( chr_dados > 100);
  chr_dados = chr_dados + 4;
  sprintf( inicial , "%d" , chr_dados);
  mensagem[0] = '\0';
  if (chr_dados<10) strcat(mensagem," ");
  strcat(mensagem , inicial);
  printf("\n** Escolha uma das opcoes abaixo :\n\n");
  printf("(P) - Retorna a sequencia de caracteres nas posicoes pares.\n");
  printf("(I) - Retorna a sequencia de caracteres nas posicoes impares.\n");
  printf("(R) - Retorna a sequencia de caracteres de forma invertida.\n");
  printf("\nAguardando escolha :\n>");

  switch ( tolower(le_caracter()) )
  {
       case 'p' :
               strcat(mensagem,"P#");
               break;
       case 'i' :
               strcat(mensagem,"I#");
               break;
       case 'r' :
               strcat(mensagem,"R#");
               break;
       default  :
               printf("\nOpcao inexistente : fim do programa\n");
               exit(1);
  }
  strcat(mensagem , dados);
  strcat(mensagem , "#");
  memcpy(end_mens , mensagem , strlen(mensagem) +1);
  printf("\nA mensagem a ser enviada sera a seguinte :\n\n");
  printf("Formato : [tamanho da mensagem][indicador de funcao]#[dados]#\n");
  printf("Mensagem: %s\n" , mensagem);
}

void main(void)
{
  //declaracao de variaveis locais
  char mens_cliente[107] , mens_servidor[100] ,tam_chr[4];
  int tam_resposta, sockfd;

  //abre socket para comunicacao com o SERVIDOR na porta 40000
  sockfd = open_client_socket(40000);
```

```
  if (sockfd == 0)
  {
     printf("Houve algum erro na abertura da conexao. Execucao suspensa.\n");
     exit(0);
  }

  for(;;)
  {
    mens_cliente[0] = mens_servidor[0] = tam_chr[0] = '\0';

    //prepara mensagem
    prepara_mensagem(mens_cliente);

    printf("\nTecle algo para enviar a mensagem e obter o resultado.\n>");
    le_caracter();

    system("clear");

    //envia mensagem para o SERVIDOR
    printf("Enviando mensagem para o SERVIDOR...\n");
    sleep(3);
    write(sockfd , mens_cliente , strlen(mens_cliente)+1);

    //recebe resposta do SERVIDOR
    printf("Aguardando resposta do SERVIDOR...\n");
    sleep(3);
    read(sockfd , tam_chr , 2);  //le tamanho da mensagem de resposta
    tam_resposta = atoi(tam_chr);
    read(sockfd , mens_servidor , tam_resposta); //le resposta

    printf("\nMensagem enviada :\n%s\n" , mens_cliente);
    printf("\nResultado obtido :\n%s\n" , mens_servidor);
    printf("Tecle T para terminar ou qualquer outra tecla para continuar.\n>");
    if (tolower(le_caracter()) == 't') //envia mensagem de TERMINA
    {
       mens_cliente[0] = '\0';
       strcat(mens_cliente , " 2T");
       write(sockfd , mens_cliente , 4);
       printf("\nFIM DO EXEMPLO\n");
       close(sockfd); // fecha conexao com o SERVIDOR
       exit(1);
    }
  }
}
```

D10 – *servidor.c*

```
#include <ctype.h>
#include "s_socket.h"

void main(void)
{
  //declaracao de variaveis locais
  char mens_cliente[107], mens_servidor[104], temp[100], tam_chr[4], funcao;
  int tam_pedido, contador, index_servidor, sockfd ;

  //abre socket para comunicacao com os CLIENTES na porta 40000
  system("clear");
  printf("\n%62s\n","***********SERVIDOR EXECUTANDO************");
  sockfd = open_server_socket(40000);

  for(;;)
  {
    mens_cliente[0] = mens_servidor[0] = tam_chr[0] = funcao ='\0';
    index_servidor = 0;

    //le pedido do CLIENTE
    read(sockfd , tam_chr , 2);
    tam_pedido = atoi(tam_chr);
```

```
    read(sockfd , mens_cliente , tam_pedido);

    //reconhece caracter de funcao
    funcao = mens_cliente[0];
    //toma procedimento de acordo com o requisitado
    if (funcao == 'P')
    {
       for (contador=0 ; contador<=(strlen(mens_cliente)-4) ; contador+=2)
       {
         temp[index_servidor] = mens_cliente[contador+2];
         index_servidor++;
       }
    }
    else if (funcao == 'I')
    {
        for (contador=1 ; contador<=(strlen(mens_cliente)-4) ; contador+=2)
        {
          temp[index_servidor] = mens_cliente[contador+2];
          index_servidor++;
        }
    }
    else if (funcao == 'R')
    {
       for (contador=0 ; contador<=(strlen(mens_cliente)-4) ; contador++)
       {
         temp[index_servidor]=mens_cliente[(strlen(mens_cliente)-2-contador)];
         index_servidor++;
       }
    }
    else if (funcao == 'T')
    {
       close(sockfd);
       exit(1);
    }
    temp[index_servidor] = '\0';
    sprintf(tam_chr , "%d" , (strlen(temp)+1) );
    if ( (strlen(temp)+1) < 10) strcat(mens_servidor , " ");
    strcat(mens_servidor , tam_chr);
    strcat( mens_servidor , temp);

    //retorna resposta para o CLIENTE
    write(sockfd, mens_servidor , (strlen(mens_servidor)+1) );
  }
}
```

D11 – *hostname.h*

```
#define SERV_HOST_NAME          "magma.del.ufrj.br"
```